Linux Kernel: Scenarios and Analyses

毛德操　胡希明　著

（上册）

LINUX 内核源代码情景分析

```
419     * This should really return information about whether
420     * we should do bottom half handling etc. Right now we
421     * end up _always_ checking the bottom half, which is a
422     * waste of time and is not what some drivers would
423     * prefer.
424     */
425     int handle_IRQ_event(unsigned int irq, struct pt_regs * regs,
          struct irqaction * action)
426     {
427        int      status;
428        int      cpu = smp_processor_id( );
429
430        irq_enter(cpu, irq);
431
432        status = 1;/* Forçe the "do bottom halves" bit */
433
434        if(!(action->flags & SA_INTERRUPT))
435              __sti( );
436
437        do{
438              status |= action->flags;
439              action->handler(irq, action->dev_id, regs);
440              action = action->next;
441        }while (action);
442        if(status & SA_SAMPLE_RANDOM)
443              add_interrupt_randomness(irq);
444        __cli( );
445
446       irq_exit(cpu, irq);
447
448       return status;
449     }
```



浙江大學出版社

目 录

第7章

基于socket的进程间通信

7.1 系统调用socket( )

相对于传统的Unix IPC，“插口”，即socket（有些资料中也称“套接字”），是更为一般的进程间通信机制。它既适用于同一台计算机上的进程间通信，也适用于网络环境的进程间通信，并且是当今所有网络操作系统必不可少的基础功能。本章侧重从同一计算机上的进程间通信的角度介绍socket在Linux内核中的实现。至于它在网络环境中的推广，则因分量太重、篇幅太大，只好留作另一本书的内容。

在Unix的发展史中，AT&T的贝尔实验室与加州大学伯克利分校的伯克利软件发布中心（BSD）可以说是两大主力。当AT&T致力于改进传统的Unix进程间通信功能，从而形成了一整套SysV IPC机制的同时，BSD也在设法对其加以改进。与此同时，BSD又最早将计算机网络的通信规程，特别是当时正在成形的TCP/IP规程，实现到Unix的内核中去。所以，对于当时的BSD来说，很自然地会把二者结合在一起考虑，把同一计算机（或曰网络“节点”）上的进程间通信纳入更广的、网络范围的进程间通信范畴，从而设计出一种更为一般化的进程间通信机制。这种努力的结果就是socket机制，这一机制其实是命名管道在计算机网络环境下的实现和推广。

如果比较一下AT&T和BSD各自在这方面的努力，就可以看出，AT&T是有系统地、全面地对传统的Unix IPC加以改进。例如，针对在某些应用中（传统Unix IPC的）效率不够高的缺点，设计和开发了共享内存机制；针对缺乏进程间同步手段的问题，又设计和开发了用户空间的信号量机制。另外，管道要占用打开文件号，便引入了“键值”的概念，从而避开了使用打开文件号。但是，AT&T的眼光始终只盯着单一计算机中的进程间通信。而BSD则正好相反，它对传统Unix进程间通信的改进并不是有系统的和全面的，可是它的眼光却跳出了单机的范围。虽然socket机制在单机范围内与AT&T的报文传递机制在概念上极为相似，但是却更为一般化，从而为后来网络技术的蓬勃发展做好了技术准备。可以说，SysV IPC和socket是互相补充而不是各搞一套。

至于Linux，则很自然地兼收并蓄，把二者都继承下来了。

顾名思义，一个socket就好像一个通信线的插口。只要通信的双方都有插口，并且两个插口之间有通信线路相连接，就可以互相通信了。从概念上说，socket与管道其实并无多大区别，如果把两个

插口之间的"连线"比喻成"管道"，那么插口就相当于管道两端的"水龙头"，而且二者都表现为已打开文件。不同的是管道两端只能在同一台计算机上，而通过虚拟的"通信线路"相连接的两个插口却可以分别存在于计算机网络中的不同节点上。当然，尽管二者在概念上相似，在具体的实现和使用上却有着很大的不同。而且，管道所传递的是无结构的字节流，而通过 socket 传递的则是有结构的报文。

一个插口在逻辑上有三个特征，或者说三个要素，那就是网域、类型以及规程。

首先是网域，它表明一个插口是用于哪一种网络，或者说哪一族网络规程的。由于各种网络对节点地址的命名方法不同，所以又称为"地址族"（address family）或"规程族"（protocol family）。例如，常数 AF_INET 表示互联网（英特网）插口，所以各节点都使用 IP 地址；而 AF_IPX 则为 Novell 的 IPX 网插口，AF_X25 为 X.25 网插口，等等。其中有个特例，那就是什么网也不是，只是在一台计算机上用于进程间通讯，BSD 为这种特例定义的域名为 AF_UNIX。后来，在 POSIX 标准里又定义了一个 AF_LOCAL，以示对别的操作系统也一视同仁。

其次为"类型"，它表明在网络中通信所遵循的模式。网络通信有两种主要模式，一种称为"有连接"或"面向连接"（connection oriented）的通信；另一种则称为"无连接"（connectionless）通信。"有连接"模式常常又称为"虚电路"（virtual circuit）模式。在这种模式中，通信的双方要先通过一定的步骤在互相之间建立起一种虚拟的连接，或者说虚拟的线路，然后再通过虚拟的连接线路进行通信。在通信的过程中，所有报文传递都保持着原来的次序，报文在网络中传输的过程中受到的不均匀延迟会在接收端得到补偿。所以所有报文之间都是有关联的，每个报文都不是孤立的。更重要的是，在这种模式中所有报文的传递都是"可靠"的，由网络中物理通信线路引入的差错会由通信规程中的应答和重发机制加以克服。同时，在这种模式中还提供了"流量控制"的手段。从用户的角度来看，"有连接"类型的插口对报文的传递作出了承诺。如果一个进程通过系统调用在一个"有连接"插口上发送一个报文，那么，只要进程从这次系统调用正常返回，就说明该报文已经被递交到了接收方的插口（但接收方进程未必已经读取这个报文）。"无连接"模式就不同了。"无连接"模式常常又称为"数据包"（datagram）模式，也称"面向报文"的通信模式（message oriented）。在"无连接"模式中并不需要事先在双方之间建立起"虚电路"，而直接就可以发送或接收报文，但是每个报文都是孤立的，其正确性也没有保证，甚至可能丢失。如果两个报文穿越网络时走过了不同的路径，或者甚至相同的路径，但是受到了不一致的延迟，那么它们在接收端的次序就可能与发送端的次序不同。所以，由"无连接"模式的插口所提供的报文传递是不可靠的，它对用户作出的承诺只是"尽力传递"，但却是没有保证的。此外，在"无连接"模式中也没有"流量控制"手段。如果一个进程通过系统调用在一个"无连接"插口上发送一个报文，那么当进程从这次系统调用正常返回时只是说明该插口将会例行公事地将报文传递到接收方，但并不表示这报文已经到达了接收方的插口。从另一个角度，还可以说，"有连接"插口的报文传递是同步的，而"无连接"插口的报文传递则是异步的。

最后是"规程"，它表明具体的网络规程。一般来说，网域和类型结合在一起大体上就确定了适用的规程。例如，要是网域为 AF_INET，而类型为"无连接"，则规程基本上就是 UDP 了。但是，在有些情况下还可能会有别的选择，此时就由它来进一步明确具体的规程。

其实，插口的这三个特征是互相联系的，归根结底就是反映了一个插口所运行的（网络）规程。由于在每个插口的后面都隐藏着网络规程，对插口的比较详尽的讨论就势必要涉及到计算机网络这个课题。要在本书中谈论计算机网络，并进而分析 Linux 内核中有关网络规程的代码是不现实的，因为那本身就已经足够构成另一本书的材料了。所以，我们在本书中将只讨论 Unix 域的插口，也就是用于同一计算机中进程间通信的插口，隐藏在这种插口后面的规程是"没有规程"。

Unix 域的插口同样也分两种模式，但是二者都提供可靠的报文传递。在网络环境下，不可靠性是由网络的基础设施（如物理线路）引起的，两种不同模式实际上反映了对待物理介质的不可靠性的不同态度和策略。“有连接”模式采取了“大包大揽”的态度，企图在不可靠物理介质的基础上构筑起一层可靠的报文传递机制。而“无连接”模式则采取了“矛盾上交”的态度，说“既然物理介质不可靠，那我也没有办法，只能有什么给你什么”，让用户自己去设法克服或避免由此而引起的问题。可是，Unix 域的插口既然只提供同一台计算机上的进程间通信，而根本就不涉及网络设施，其物理介质实际上就是内存，那就根本不存在由物理介质引入的不可靠性。不过，尽管 Unix 域插口的两种模式都提供可靠的报文传递，二者还是有区别的，读完本节以后就会清楚这些区别。

像对 Sys V IPC 操作一样，Linux 内核为所有与 socket 有关的操作提供一个统一的系统调用入口，但是在用户程序界面上则通过 C 语言程序库 c.lib 提供诸多库函数，看起来就好像都是独立的系统调用一样。内核中为 socket 设置的总入口为 sys_socketcall( )，其代码在 net/socket.c 中：

```
1512	/*
1513	 *	System call vectors.
1514	 *
1515	 *	Argument checking cleaned up. Saved 20% in size.
1516	 *	This function doesn't need to set the kernel lock because
1517	 *	it is set by the callees.
1518	 */
1519
1520	asmlinkage long sys_socketcall(int call, unsigned long *args)
1521	{
1522		unsigned long a[6];
1523		unsigned long a0,a1;
1524		int err;
1525
1526		if(call<1||call>SYS_RECVMSG)
1527			return -EINVAL;
1528
1529		/* copy_from_user should be SMP safe. */
1530		if (copy_from_user(a, args, nargs[call]))
1531			return -EFAULT;
1532
1533		a0=a[0];
1534		a1=a[1];
1535
1536		switch(call)
1537		{
1538			case SYS_SOCKET:
1539				err = sys_socket(a0,a1,a[2]);
1540				break;
1541			case SYS_BIND:
1542				err = sys_bind(a0,(struct sockaddr *)a1, a[2]);
1543				break;
1544			case SYS_CONNECT:
```

```
1545                err = sys_connect(a0, (struct sockaddr *)a1, a[2]);
1546                break;
1547            case SYS_LISTEN:
1548                err = sys_listen(a0,a1);
1549                break;
1550            case SYS_ACCEPT:
1551                err = sys_accept(a0,(struct sockaddr *)a1, (int *)a[2]);
1552                break;
1553            case SYS_GETSOCKNAME:
1554                err = sys_getsockname(a0,(struct sockaddr *)a1, (int *)a[2]);
1555                break;
1556            case SYS_GETPEERNAME:
1557                err = sys_getpeername(a0, (struct sockaddr *)a1, (int *)a[2]);
1558                break;
1559            case SYS_SOCKETPAIR:
1560                err = sys_socketpair(a0,a1, a[2], (int *)a[3]);
1561                break;
1562            case SYS_SEND:
1563                err = sys_send(a0, (void *)a1, a[2], a[3]);
1564                break;
1565            case SYS_SENDTO:
1566                err = sys_sendto(a0,(void *)a1, a[2], a[3],
1567                         (struct sockaddr *)a[4], a[5]);
1568                break;
1569            case SYS_RECV:
1570                err = sys_recv(a0, (void *)a1, a[2], a[3]);
1571                break;
1572            case SYS_RECVFROM:
1573                err = sys_recvfrom(a0, (void *)a1, a[2], a[3],
1574                           (struct sockaddr *)a[4], (int *)a[5]);
1575                break;
1576            case SYS_SHUTDOWN:
1577                err = sys_shutdown(a0,a1);
1578                break;
1579            case SYS_SETSOCKOPT:
1580                err = sys_setsockopt(a0, a1, a[2], (char *)a[3], a[4]);
1581                break;
1582            case SYS_GETSOCKOPT:
1583                err = sys_getsockopt(a0, a1, a[2], (char *)a[3], (int *)a[4]);
1584                break;
1585            case SYS_SENDMSG:
1586                err = sys_sendmsg(a0, (struct msghdr *) a1, a[2]);
1587                break;
1588            case SYS_RECVMSG:
1589                err = sys_recvmsg(a0, (struct msghdr *) a1, a[2]);
1590                break;
1591            default:
1592                err = -EINVAL;
```

```
1593                break;
1594        }
1595        return err;
1596    }
```

函数的第一个参数 call 即为具体的操作码，而参数 args 为指向一个数组的指针。根据具体操作码的不同，需要从用户空间复制参数的数量也不同。为了根据操作码确定需要从用户空间复制的字节数，代码 socket.c 中定义了一个数组 nargs[ ]:

```
1505    /* Argument list sizes for sys_socketcall */
1506    #define AL(x) ((x) * sizeof(unsigned long))
1507    static unsigned char nargs[18]={AL(0),AL(3),AL(3),AL(3),AL(2),AL(3),
1508                    AL(3),AL(3),AL(4),AL(4),AL(4),AL(6),
1509                    AL(6),AL(2),AL(5),AL(5),AL(3),AL(3)};
1510    #undef AL
```

至于操作码，则是在 include/linux/net.h 中定义的：

```
30  #define SYS_SOCKET      1       /* sys_socket(2)        */
31  #define SYS_BIND        2       /* sys_bind(2)          */
32  #define SYS_CONNECT     3       /* sys_connect(2)       */
33  #define SYS_LISTEN      4       /* sys_listen(2)        */
34  #define SYS_ACCEPT      5       /* sys_accept(2)        */
35  #define SYS_GETSOCKNAME 6       /* sys_getsockname(2)   */
36  #define SYS_GETPEERNAME 7       /* sys_getpeername(2)   */
37  #define SYS_SOCKETPAIR  8       /* sys_socketpair(2)    */
38  #define SYS_SEND        9       /* sys_send(2)          */
39  #define SYS_RECV        10      /* sys_recv(2)          */
40  #define SYS_SENDTO      11      /* sys_sendto(2)        */
41  #define SYS_RECVFROM    12      /* sys_recvfrom(2)      */
42  #define SYS_SHUTDOWN    13      /* sys_shutdown(2)      */
43  #define SYS_SETSOCKOPT  14      /* sys_setsockopt(2)    */
44  #define SYS_GETSOCKOPT  15      /* sys_getsockopt(2)    */
45  #define SYS_SENDMSG     16      /* sys_sendmsg(2)       */
46  #define SYS_RECVMSG     17      /* sys_recvmsg(2)       */
```

注释中括号里的“2”表示所述的函数为系统调用。

我们先把这些操作码（以及相应的函数）分一下类，然后说明它们的用途，最后再来看它们是怎样实现的。

这些操作码（函数）大体上可分为五类。

7.1.1　插口的创建与撤除

属于这一类的操作码有：

- SYS_SOCKET：创建一个插口，其用户程序界面（由 libc 提供，下同）为：

```
int socket (int domain, int type, int protocol);
```

这里的三个参数即为前面所述插口的三个要素。不过，通常第三个参数 protocol 为 0，因为一般来说前两个参数确定以后，具体的规程也就定了，所以用 0 表示默认由系统根据前两个参数确定的规程，只有在比较特殊的应用中才需要指定具体的规程。插口创建成功以后返回一个正整数，实际上是一个打开文件号。但这个打开文件号是与一个代表插口的数据结构相联系的，并不是与磁盘上的某个文件相联系。

- SYS_BIND：将代表着一个插口的打开文件号与某一个域中的可寻址实体或“插口地址”相结合。例如，在互联网中的可寻址实体为网中以 IP 地址区分的节点，而在同一节点上又有若干不同的“传输层”收发口（逻辑意义上），所以此时的结合就是与 IP 地址加收发口逻辑编号的结合。在 Unix 域中，此种可寻址实体是一个文件名，所以就成了与文件名的结合。用户程序界面为：

```
int bind(int sockfd , struct sockaddr *my_addr, socklen_t addrlen);
```

这里的 sockfd 即为 socket( )所返回的打开文件号。在 bind 到某个地址或文件名之前，虽然插口已经建立但却无从寻访。如果说 socket( )好像是安装了一个电话机的话，那么 bind( )就好像为它指定了一个电话号码。

- SYS_SOCKETPAIR：创建一对已经互相连接的无名插口，概念上类似于 pipe( )。用户程序界面为：

```
int socketpair (int domain, int type , int protocal , int sv[2]);
```

前面三个参数与 socket( )相同，数组 sv[ ]则用来返回创建后的一对打开文件号。

- SYS_ SHUTDOWN：部分或完全关闭一个插口，用户程序界面为：

```
int shutdown (int s , int how);
```

参数 s 为插口的打开文件号，参数 how 为 0 表示不再允许接收，1 表示不再允许发送，2 则表示接收和发送都不再允许。由于插口体现为特殊的打开文件，所以也可以用 close( )来关闭。

7.1.2 插口间连接的建立

“有连接”模式通信的双方是不对称的，可以说天生就是 client/server 的关系。而连接的建立则必须经过一定的步骤：

- SYS_LISTEN：作为 server 的一方首先要通过 listen( )向内核“挂号”。只有挂了号的插口才会被内核视为 server 一侧的插口并允许它接受来自 client 一侧的连接请求。用户程序界面为：

```
int listen(int s, int backlog);
```

参数 s 即为插口的打开文件号。当连接请求到来，而一时得不到接受时，就暂时进入一个队列，在那里等待 server 方的接受。这里的参数 backlog 就规定了这个队列的最大长度。

- SYS_CONNECT：在“有连接”模式中，client 一方要通过 connect( )向已经挂了号的 server 方插口请求连接。用户程序界面为：

```
int connect (int sockfd, const struct sockaddr *serv addr, socklen_t addrlen);
```

参数 sockfd 为 client 一侧插口的打开文件号，serv_addr 则指向一个表示 server 方插口地址的数据结构。当然，这个地址必须已经由 server 方通过 bind( )指定给 server 方的插口。由于有些网域中的地址可能不是固定长度的，所以还要有个参数 addrlen 来指明其长度。

对 connect( )的调用一般要到 server 方接受了连接或者出错时才返回。

- SYS_ACCEPT：server 方通过 accept( )接受或等待接受到来的（或已经在队列中的）第一个连接请求。用户程序界面为：

```
int  accept(int s, struct sockaddr  *addr , socklen_t  *addrlen);
```

参数 s 为一个已经挂了号的 server 方插口（打开文件号），参数 addr 与 addrlen 则用来返回连接请求的来源，也就是 client 方插口的地址。对 accept( )调用要到有连接请求到来而建立起连接，或者发生了出错时才返回。

特别要说明的是，这个函数返回一个新的打开文件号。这个新的打开文件号就代表着已经建立起连接的 server 方插口。而原来的插口 s，则保持原样不变，又可以用来接受新的（其他的）连接请求。这个插口就好像是下蛋的鸡，每接受一个连接请求就生下一个“蛋”，那就是已经建立起连接的插口。在典型的应用中，server 进程在通过 accept( )接受了一个连接请求，从而与一个 client 进程建立起一个连接以后，就会通过 fork( )创建出一个子进程。然后，子进程将作为“种籽”的 server 方插口关闭，而使用新的插口与 client 进程通信并为之提供服务。而父进程则把新的插口关闭，并且再一次调用 accept( )，通过“种籽”插口来接受新的连接请求。这就是典型的 client/server 运行模式。

7.1.3 “有连接”模式的报文发送与接收

“有连接”模式的插口一定要在 client 和 server 双方建立起连接以后才能用于通信。由于插口在用户界面上表现为已打开文件，并且“有连接”模式的通信既是可靠的又保持原有的次序，所以可以把传递的信息看作有序的字节流，从而可以用普通的 read( )和 write( )系统调用，像读/写普通文件一样地来接收和发送信息。除此之外，也可以用专门为插口而设的三对库函数之一来接收和发送，即 recv( )/send( )、recvfrom( )/send_to( )以及 recvmsg( )/sendmsg( )。不过，这些库函数主要用于“无连接”模式，所以我们将它们与“无连接”模式的发送与接收放在一起介绍。这里还要指出，无论是“有连接”还是“无连接”模式，插口都是双向的，并且就发送和接收而言双方是对称的。

7.1.4 “无连接”模式的报文发送与接收

顾名思义，“无连接”模式的插口不需要事先建立连接，插口一经创建马上就可以发送或接收报文。另一方面，由于“无连接”模式的通信既不是可靠的，也不一定保持原有的次序，所以不宜用 read( ) 和 write( )像对待一个有序字节流那样使用“无连接”模式的插口，而要用专门设置的三组库函数来进行。与文件操作 read( )、write( )相比，这三组函数的特点是它们都保留报文的边界，有关这一点读者在后面看了源代码以后就清楚了。

- SYS_SENDTO：如前所述，“无连接”模式的插口一经建立（并且 bind 到一个插口地址）以后就可以进行通信，而无需先建立连接。当然，此时对所发送的每个报文都要提供对方的地址（在“无连接”模式中，每个报文都是独立的），所以，sendto( )的用户程序界面为：

```
int  sendto( int  s ,  const void * msg ,  size_t  len,  int  flags,
                    const struct sockaddr  *to,  socklen_t  tolen);
```

显然，这个函数是专为“无连接”模式的报文发送而设置的，参数 to 指向对方的地址即 sockaddr 数据结构，而 tolen 则为地址的长度。虽然这个函数的界面是专为“无连接”模式设计的，但是也可以用这个函数在“有连接”模式的插口上发送报文，不过此时应将参数 to 设置成 NULL，而将 tolen 设置成 0。

在内核中，操作 SYS_SENDTO 是由 sys_sendto( )实现的。

- SYS_SEND：从用户程序界面来看，send( )似乎主要是为“有连接”模式设计的：

```
int  send(int  s,  const void  *msg ,  size_t  len ,  int  flags);
```

与 sendto( )的界面相比，这里没有提供对方的地址。在“有连接”模式中，连接已经事先建立好，当然不需要每次都提供对方地址了。但是，即使在“无连接”模式中，当准备接连向同一目标发送很多个报文时，每次都要提供对方的地址。这样做既麻烦又降低了效率（每次都要从用户空间把地址复制到内核中）。是不是可以简化一下呢？例如，是否可以先“预设”一个双方地址，随后就采用 send( )来发送，而不必每次都重复地提供相同的地址。事实正是这样，对于“无连接”模式的插口，可以用 connect( )先设置一个对方地址，然后再用 send( )发送报文，而实际上每次都使用预先设置好的对方地址。但是要注意，在“无连接”模式中使用 connect( )与在“有连接”模式中使用 connect( )有本质的区别。在“无连接”模式中，connect( )的作用只是让内核为“本地”插口记下预设的对方地址，而并不涉及与对方之间控制报文的往返。以后则在发送的每个报文头部附上这个地址，以指明报文的目的地。至于在“有连接”模式中的 connect( )，则实际向对方发送一个请求连接的控制报文（指在网络环境下），并等待对方的响应。连接建立了以后，随同每个报文发送的可以只是一个连接号，而不一定要包括对方地址。所以，虽然在形式上两种模式都可以使用 connect( )，并且有时把经过 connect( )预设好对方地址的“无连接”插口也说成是处于“已连接状态”，但实质上是完全不同的。读者在资料中碰到此类词句时要注意根据上下文加以区分。

在内核中，SYS_SEND 是由 sys_send( )实现的，而 sys_send( )又是由 sys_sendto( )实现的，代码见 net/socket.c：

[sys_socketcall( ) > sys_send( )]

```
1207    asmlinkage long sys_send(int fd, void * buff, size_t len, unsigned flags)
1208    {
1209        return sys_sendto(fd, buff, len, flags, NULL, 0);
1210    }
```

- SYS_SENDMSG：库函数 sendmsg( )是前两个函数的推广与加强，这是三个函数中最复杂的，其用户程序界面为：

```
int  sendmsg(int  s ,  const struct msghdr  *msg ,  int  flags);
```

参数 msg 指向一个 msghdr 数据结构，定义于 socket.h：

```
33    struct msghdr {
34        void        *msg_name;    /* Socket name          */
35        int         msg_namelen;  /* Length of name       */
36        struct iovec  *msg_iov;     /* Data blocks          */
37        __kernel_size_t    msg_iovlen; /* Number of blocks     */
```

```
38      void      *msg_control;    /*Per protocol magic(eg BSD file
                                              descriptor passing)*/
39      __kernel_size_t     msg_controllen; /* Length of cmsg list */
40      unsigned     msg_flags;
41  };
```

每个 msghdr 数据结构都代表着一个报文。在 msghdr 结构中，msg_name 为对方的插口地址（或者也可以称作插口名）。而 msg_iov 则指向一个结构数组，该数组中的每一个元素都是一块数据，这样一个报文的内容就可以分散在若干个互不连续的缓冲区中，而在逻辑上却连在一起，在网络环境下这是很有好处的。还有，msg_control 和 msg_controllen 的作用是传递控制信息，在 Unix 域中用来在进程间传递访问权限，还可以用来传递"打开文件描述体"。这些以后再介绍。

由于每个报文都有个对方地址，这显然适合于"无连接"模式。但是，像 sendto( )一样，它也可以用于"有连接"模式中，不过要将 msg_name 设成 NULL，msg_namelen 设成 0。

- SYS_RECV、SYS_RECVFROM、SYS_RECVMSG：这三个操作都是用来从某个插口 s 接收报文的，并且分别与前列用来发送报文的操作相对应，所以我们把它们合并在一起叙述。完成这些操作的库函数为：

```
int  recv(int  s,  void  *buf ,  size_t  len ,  int  flags);
int  recvfrom(int  s,  void  *buf ,  size_t  len ,  int  flags,
                       struct sockaddr  *from ,  socklen_t  *fromlen);
int  recvmsg (int  s ,  struct msghdr  *msg ,  int  flags);
```

与发送报文的库函数类似，recv( )通常用于"有连接"模式的插口，或者虽然是"无连接"模式的插口，但却已经用 connect( )预设了对方地址。函数 recvfrom( )的接收是全方位的，参数 from 并不是用来指定接收来自哪一个远方插口的报文，而是用来在接收到一个报文时指明报文的来源。所以 from 所指向的数据结构只是一个用于返回报文来源的缓冲区，而 fromlen 所指则为该缓冲区的大小。在内核中，SYS_RECV 由 sys_recv( )实现，SYS_RECVFROM 则由 sys_recvfrom( )实现。但是，就像 sys_send( )与 sys_sendto( )之间的关系一样，sys_recv( )也是通过 sys_recvfrom( )实现的，代码见 net/socket.c：

[sys_socketcall( ) > sys_recv( )]

```
1258    asmlinkage long sys_recv(int fd, void * ubuf, size_t size, unsigned flags)
1259    {
1260        return sys_recvfrom(fd, ubuf, size, flags, NULL, NULL);
1261    }
```

也就是说，如果把 recvfrom( )的参数 from 和 fromlen 设置成 NULL，就跟 recv( )一样了。函数 recvmsg( )则与 sendmsg( )相对应，也具有将一个报文的内容分散在若干个缓冲区内的功能，并且具有接收控制信息的功能。

7.1.5 控制以及辅助性的操作

- SYS_GETSOCKNAME：库函数 getsockname( )用来获取为一个插口 s 指定（bind）的地址或名字：

```
int  getsockname(int  s,  struct sockaddr  *name ,  socklen_t  *namelen);
```

- SYS_GETPEERNAME："有连接"模式的插口，不管是 client 方还是 server 方，一旦建立起连接以后就有了一个"对方"。库函数 getpeername( )用来获取插口 s 的对方插口的地址（或名字）：

```
int  getpeername(int  s,  struct  sockaddr  * name ,  socklen_t  * namelen);
```

- SYS_SETSOCKOPT、SYS_GETSOCKOPT：库函数 setsockopt( )和 getsockopt( )用来设置和读取一个插口 s 所实行规程中的一些可选项，由于可选项都是在网络环境下运用的，而本书只讲述 Unix 域的插口，所以我们在这里并不关心这两个函数。

有了上面这些预备知识以后，我们就可以从下一节开始介绍 socket 通信机制在 Unix 域中的实现了。为了往后阅读的方便，此处先给出利用插口实现进程间通信的流程示意图（图 7.1）。

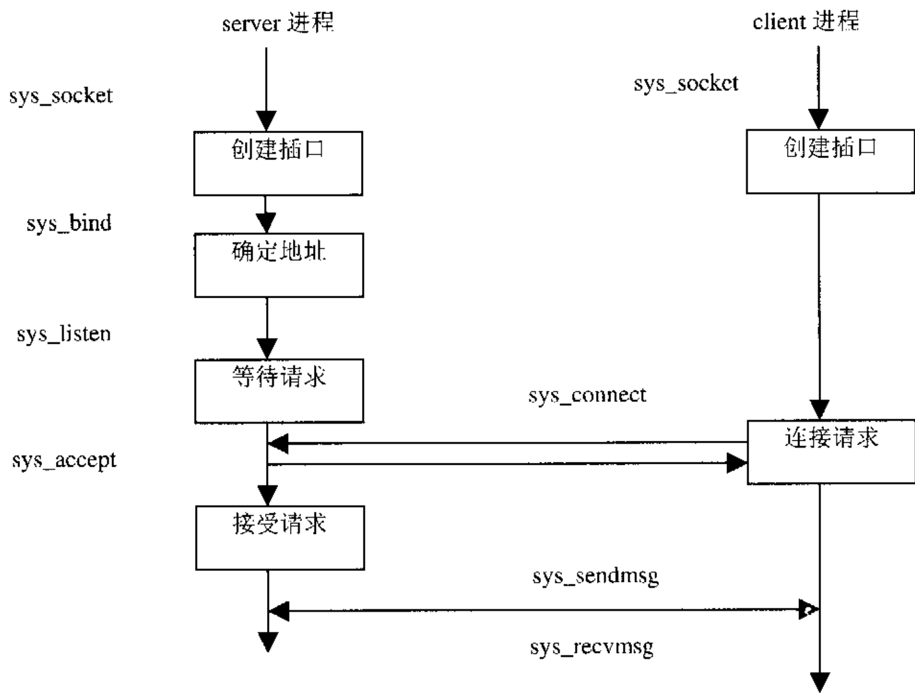


图 7.1　插口通信流程示意图

7.2　函数 sys_socket( )——创建插口

操作 SYS_SOCKET 是由 sys_socket( )实现的，其代码在 net/socket.c 中：

[sys_socketcall( ) > sys_socket( )]

```
889     asmlinkage long sys_socket(int family, int type, int protocol)
890     {
891         int retval;
892         struct socket *sock;
893 
894         retval = sock_create(family, type, protocol, &sock);
895         if (retval < 0)
896             goto out;
897 
898         retval = sock_map_fd(sock);
899         if (retval < 0)
900             goto out_release;
901 
902     out:
903         /* It may be already another descriptor 8) Not kernel problem. */
904         return retval;
905 
906     out_release:
907         sock release(sock);
908         return retval;
909     }
```

前面说过，插口对于用户程序而言就是特殊的已打开文件。内核中为插口定义了一种特殊的文件类型，形成一种特殊的文件系统 sockfs，定义于 net/socket.c 中：

```
301     static struct vfsmount *sock_mnt;
302     static DECLARE_FSTYPE(sock_fs_type, "sockfs", sockfs_read_super,
303         FS_NOMOUNT|FS_SINGLE);
```

在系统初始化时，要通过 kern_mount( )安装这个文件系统。安装时有个作为连接件的 vfsmount 数据结构，这个结构的地址就保存在一个全局的指针 sock_mnt 中。所谓创建一个插口，就是在 sockfs 文件系统中创建一个特殊文件，或者说一个节点，并建立起为实现插口功能所需的一整套数据结构。所以首先是建立一个 socket 数据结构，然后将其“映射”到一个已打开文件中。函数 sock_create( )的代码在同一文件（socket.c）中。这段代码由于比较长，我们分段来看：

[sys_socketcall( ) > sys_socket( ) > sock_create( )]

```
814     int sock_create(int family, int type, int protocol, struct socket **res)
815     {
816         int i;
817         struct socket *sock;
818 
819         /*
820          *  Check protocol is in range
```

```
821         */
822         if(family<0 || family>=NPROTO)
823             return -EAFNOSUPPORT;
824 
825         /* Compatibility.
826 
827            This uglymoron is moved from INET layer to here to avoid
828            deadlock in module load.
829          */
830         if (family == PF_INET && type == SOCK_PACKET) {
831             static int warned;
832             if (!warned) {
833                 warned = 1;
834                 printk(KERN_INFO "%s uses obsolete (PF_INET,SOCK_PACKET)\n",
                                                 current->comm);
835             }
836             family = PF_PACKET;
837         }
838 
839     #if defined(CONFIG_KMOD) && defined(CONFIG_NET)
840         /* Attempt to load a protocol module if the find failed.
841          *
842          * 12/09/1996 Marcin: But! this makes REALLY only sense, if the user
843          * requested real, full-featured networking support upon configuration.
844          * Otherwise module support will break!
845          */
846         if (net_families[family]==NULL)
847         {
848             char module_name[30];
849             sprintf(module_name,"net-pf-%d",family);
850             request_module(module_name);
851         }
852     #endif
853 
854         net_family_read_lock( );
855         if (net_families[family] == NULL) {
856             i = -EAFNOSUPPORT;
857             goto out;
858         }
859 
```

这一段代码的开始部分是检查和处理参数的范围。由于我们在这里只关心 Unix 域，也就是当 family 为 AF_UNIX 时的情景，所以这段代码对我们不起什么作用。接下来是一段条件编译，如果系统配置了可动态安装内核模块的功能，并且网络驱动程序是动态安装的，就要检查一下由参数 family 所指定网域的驱动程序是否已经安装，尚未安装的话就要调用 request_module( )把它安装起来。至于 Unix 域插口的驱动程序，那是内核所固有的，并非动态安装的模块。此外，就像每种文件系统都有个 file_operations 数据结构一样，每种网域，包括 Unix 域，也都有个 net_proto_family 数据结构。在系统

初始化或安装模块时，要将指向相应网域的这个数据结构的指针填入一个数组 net_families[ ]中，否则就说明系统尚不支持给定的网域。Unix 域的 net_proto_family 数据结构为 unix_family_ops，定义于 net/unix/af_unix.c 中：

```
1844    struct net_proto_family unix_family_ops = {
1845        PF_UNIX,
1846        unix_create
1847    };
```

就是说，结构中只有一个代表着 Unix 域的常数 PF_UNIX 和一个函数指针 unix_create，而 PF_UNIX 决定了指向这个数据结构的指针在 net_families[ ]中的位置。

回到 sys_socket( )中继续往下看（socket.c）：

[sys_socketcall( ) > sys_socket( ) > sock_create( )]

```
862    /*
863     * Allocate the socket and allow the family to set things up. if
864     * the protocol is 0, the family is instructed to select an appropriate
865     * default.
866     */
867
868        if (!(sock = sock_alloc( )))
869        {
870            printk(KERN_WARNING "socket: no more sockets\n");
871            i = -ENFILE;          /* Not exactly a match, but its the
872                                closest posix thing */
873            goto out;
874        }
875
876        sock->type = type;
877
878        if ((i = net_families[family]->create(sock, protocol)) < 0)
879        {
880            sock_release(sock);
881            goto out;
882        }
883
884        *res = sock;
885
886    out:
887        net_family_read_unlock( );
888        return i;
889        }
```

插口是由 socket 数据结构代表的，这种数据结构定义于 include/linux/net.h 中：

```
65	struct socket
66	{
67		socket_state		state;
68	
69		unsigned long		flags;
70		struct proto_ops	*ops;
71		struct inode		*inode;
72		struct fasync_struct *fasync_list; /* Asynchronous wake up list */
73		struct file	*file;	/* File back pointer for gc */
74		struct sock	*sk;
75		wait_queue_head_t	wait;
76	
77		short		type;
78		unsigned char		passcred;
79	};
```

结构中各个成分的用途随着代码的阅读会变得清楚起来，这里暂且只要知道有这些成分就可以了。不过我们建议读者在搞清了这些成分的用途以后再回过头来自己加上注释。

函数 sock_alloc( )分配一个 socket 数据结构并进行一些初始化，其代码在 net/socket.c 中：

[sys_socketcall( ) > sys_socket( ) > sock_create( ) > sock_alloc( )]

```
427	/**
428	 *	sock_alloc	-	allocate a socket
429	 *
430	 *	Allocate a new inode and socket object. The two are bound together
431	 *	and initialised. The socket is then returned. If we are out of inodes
432	 *	NULL is returned.
433	 */
434	
435	struct socket *sock_alloc(void)
436	{
437		struct inode * inode;
438		struct socket * sock;
439	
440		inode = get_empty_inode( );
441		if (!inode)
442			return NULL;
443	
444		inode->i_sb = sock_mnt->mnt_sb;
445		sock = socki_lookup(inode);
446	
447		inode->i_mode = S_IFSOCK|S_IRWXUGO;
448		inode->i_sock = 1;
449		inode->i_uid = current->fsuid;
450		inode->i_gid = current->fsgid;
451	
```

```
452         sock->inode = inode;
453         init waitqueue_head(&sock->wait);
454         sock->fasync_list = NULL;
455         sock->state = SS_UNCONNECTED;
456         sock->flags = 0;
457         sock->ops = NULL;
458         sock->sk = NULL;
459         sock->file = NULL;
460
461         sockets_in_use[smp processor_id( )].counter++;
462         return sock;
463     }
```

可见，取得一个 inode 结构是取得一个 socket 结构的必要条件。不仅如此，socket 结构其实只是 inode 结构中的一部分！读者在“文件系统”一章中看到过有关 inode 数据结构的说明，在 inode 结构中有一个关键性的成分 u。这是一个 union，要按具体的文件类型和格式而解释成不同的数据结构。目前 Linux 支持 20 多种不同的文件系统，因此对这个 union 有 20 多种不同的解释，而 socket 结构正是其中之一。代码中 445 行的 socki_lookup( ) 只是将 inode 结构中的这个 union 解释为 socket 结构而已，见 net/socket.c 中的代码：

[sys_socketcall( ) > sys_socket( ) > sock_alloc( ) > socki_lookup( )]

```
377     extern __inline__ struct socket *socki_lookup(struct inode *inode)
378     {
379         return &inode->u.socket_i;
380     }
```

同时，在 inode 结构中还要将 i_mode 里的 S_IFSOCK 标志位设成 1，并将 i_sock 也设成 1，以示这个 inode 结构所代表的并不是磁盘文件，而是一个插口。

分配了一个 socket 结构，并且设置好插口的类型以后，就通过由 unix_family_ops 提供的函数指针 create 调用 Unix 域的插口创建程序 unix_create( )，其代码在 af_unix.c 中：

[sys_socketcall( ) > sys_socket( ) > sock_create( ) > unix_create( )]

```
498     static int unix_create(struct socket *sock, int protocol)
499     {
500         if (protocol && protocol != PF_UNIX)
501             return -EPROTONOSUPPORT;
502
503         sock->state = SS_UNCONNECTED;
504
505         switch (sock->type) {
506         case SOCK_STREAM:
507             sock->ops = &unix_stream_ops;
508             break;
509             /*
```

```
510              * Believe it or not BSD has AF_UNIX, SOCK_RAW though
511              * nothing uses it.
512              */
513         case SOCK_RAW:
514             sock->type=SOCK_DGRAM;
515         case SOCK_DGRAM:
516             sock->ops = &unix_dgram_ops;
517             break;
518         default:
519             return -ESOCKTNOSUPPORT;
520         }
521 
522         return unix_create1(sock) ? 0 : -ENOMEM;
523     }
```

参数 protocol 是从用户空间一直传下来的，若为 0 则默认为 Unix 域。插口的初始状态设置成 SS_UNCONNECTED，“有连接”模式，即类型为 SOCK_STREAM 的插口必须在建立了连接以后才能使用。插口的类型决定了通信的模式，对于给定的网域这通常也决定了采用的规程。对插口的各种操作因规程的不同而异，所以各种规程都有其自己的一套插口操作，通过一个 proto_ops 数据结构提供有关的函数指针。根据插口类型的不同，这里把 socket 结构中的指针分别设置成 unix_stream_ops 或 unix_dgram_ops。这两个数据结构均定义于 net/unix/af_unix.c 中：

```
1804    struct proto_ops unix_stream_ops = {
1805        family:     PF_UNIX,
1806 
1807        release:    unix_release,
1808        bind:       unix_bind,
1809        connect:    unix_stream_connect,
1810        socketpair: unix_socketpair,
1811        accept:     unix_accept,
1812        getname:    unix_getname,
1813        poll:       unix_poll,
1814        ioctl:      unix_ioctl,
1815        listen:     unix_listen,
1816        shutdown:   unix_shutdown,
1817        setsockopt: sock_no_setsockopt,
1818        getsockopt: sock_no_getsockopt,
1819        sendmsg:    unix_stream_sendmsg,
1820        recvmsg:    unix_stream_recvmsg,
1821        mmap:       sock_no_mmap,
1822    };
1823 
1824    struct proto_ops unix_dgram_ops = {
1825        family:     PF_UNIX,
1826 
1827        release:    unix_release,
```

```
1828        bind:       unix_bind,
1829        connect:    unix_dgram_connect,
1830        socketpair: unix_socketpair,
1831        accept:     sock_no_accept,
1832        getname:    unix_getname,
1833        poll:       datagram_poll,
1834        ioctl:      unix_ioctl,
1835        listen:     sock_no_listen,
1836        shutdown:   unix_shutdown,
1837        setsockopt: sock_no_setsockopt,
1838        getsockopt: sock_no_getsockopt,
1839        sendmsg:    unix_dgram_sendmsg,
1840        recvmsg:    unix_dgram_recvmsg,
1841        mmap:       sock_no_mmap,
1842    };
```

对比一下上列这两个数据结构，就可以看出在无连接模式插口中对应于 accept 的函数为 sock_no_accept( )，这个函数只是返回一个出错代码－EOPNOTSUPP，表示不支持所要求的操作。同样地，也不支持 listen。同时，两种模式的 Unix 域插口都不支持 setsockopt 和 getsockopt，也不支持 mmap。不过，如前所述，对于用户程序而言，插口就是已打开文件，所以还可能通过常规的文件操作界面支持一些比较通用的操作，有些操作如 unix_poll( )、unix_ioctl( )等，并不是通过 sys_socketcall( )来调用的，而要通过普通的文件操作界面来调用。

此外，一般网域的插口类型除 SOCK_STREAM 和 SOCK_DGRAM 外还有一种 SOCK_RAW，也是无连接模式的，在 Unix 域中则等同于 SOCK_DGRAM，所以 514 行把插口类型改成 SOCK_DGRAM，注意在 514 行下面没有 break 语句。

最后，unix_create( )调用 unix_create1( )，进一步完成创建插口的任务（af_unix.c）：

[sys_socketcall( ) > sys_socket( ) > sock_create( ) > unix_create( ) > unix_create1( )]

```
463     static struct sock * unix_create1(struct socket *sock)
464     {
465         struct sock *sk;
466 
467         if (atomic_read(&unix_nr_socks) >= 2*files_stat.max_files)
468             return NULL;
469 
470         MOD_INC_USE_COUNT;
471         sk = sk_alloc(PF_UNIX, GFP_KERNEL, 1);
472         if (!sk) {
473             MOD_DEC_USE_COUNT;
474             return NULL;
475         }
476 
477         atomic_inc(&unix_nr_socks);
478 
479         sock_init_data(sock, sk);
```

```
480
481         sk->write_space     =   unix_write_space;
482
483         sk->max_ack_backlog = sysctl_unix_max_dgram_qlen;
484         sk->destruct = unix_sock_destructor;
485         sk->protinfo.af_unix.dentry=NULL;
486         sk->protinfo.af_unix.mnt=NULL;
487         sk->protinfo.af_unix.lock = RW_LOCK_UNLOCKED;
488         atomic_set(&sk->protinfo.af_unix.inflight, 0);
489         init_MUTEX(&sk->protinfo.af_unix.readsem);/* single task reading lock */
490         init_waitqueue_head(&sk->protinfo.af_unix.peer_wait);
491         sk->protinfo.af_unix.list=NULL;
492         unix_insert_socket(&unix_sockets_unbound, sk);
493
494         return sk;
495     }
```

每个插口都有个 socket 数据结构，同时又有个 sock 数据结构，后者可以说是对前者的一种扩充，两者间是一对一的对应关系。在 socket 结构中有个指针 sk 指向其对应的 sock 结构，而在 sock 结构中则有个指针 socket 指向其对应的 socket 结构，所以二者可以说是同一个东西的两个侧面。其中 sock 结构是一种相当大的数据结构，其定义有 180 多行，我们不在这里列出了，读者可以自己在 include/linux/net/sock.h 中找到它的定义。在这里，我们只是随着代码的进展对有关的成分作一些介绍。另一方面，sock 结构是内核中常常要动态分配使用的，所以内核中为此专设了一个队列，通过 slab 机制来管理这种数据结构的缓冲区。分配了 sock 结构以后就是它的初始化，sock_int_data( )的代码也在 sock.c 中：

[sys_socketcall( ) > sys_socket( ) > sock_create( ) > unix_create( )
 > unix_create1( ) > sock_init_data( )]

```
1117    void sock_init_data(struct socket *sock, struct sock *sk)
1118    {
1119        skb_queue_head_init(&sk->receive_queue);
1120        skb_queue_head_init(&sk->write_queue);
1121        skb_queue_head_init(&sk->error_queue);
1122
1123        init_timer(&sk->timer);
1124
1125        sk->allocation  =   GFP_KERNEL;
1126        sk->rcvbuf  =   sysctl_rmem_default;
1127        sk->sndbuf  =   sysctl_wmem_default;
1128        sk->state   =   TCP_CLOSE;
1129        sk->zapped  =   1;
1130        sk->socket  =   sock;
1131
1132        if(sock)
1133        {
```

```
1134            sk->type     =   sock->type;
1135            sk->sleep    =   &sock->wait;
1136            sock->sk     =   sk;
1137        } else
1138            sk->sleep    =   NULL;
1139
1140        sk->dst_lock          =   RW_LOCK_UNLOCKED;
1141        sk->callback_lock     =   RW_LOCK_UNLOCKED;
1142
1143        sk->state_change      =   sock_def_wakeup;
1144        sk->data_ready        =   sock_def_readable;
1145        sk->write_space       =   sock_def_write space;
1146        sk->error_report      =   sock def_error_report;
1147        sk->destruct                =       sock_def_destruct;
1148
1149        sk->peercred.pid      =   0;
1150        sk->peercred.uid      =   -1;
1151        sk->peercred.gid      =   -1;
1152        sk->rcvlowat          =   1;
1153        sk->rcvtimeo          =   MAX_SCHEDULE TIMEOUT;
1154        sk->sndtimeo          =   MAX_SCHEDULE_TIMEOUT;
1155
1156        atomic_set(&sk->refcnt, 1);
1157    }
```

在 sock 结构中有几个双向链队列，其中最重要的就是 receive_queue 和 write_queue，而 error_queue 则仅在网络环境下才会用到。这几个队列并不采用通用的队列头结构 list_head，而专门定义了一种 sk_buff_head 数据结构，定义于 include/linux/skbuff.h 中：

```
51    struct sk_buff_head {
52        /* These two members must be first. */
53        struct sk_buff  * next;
54        struct sk_buff  * prev;
55
56        __u32       qlen;
57        spinlock t  lock;
58    };
```

排在队列中的都是 sk_buff 数据结构。在网络环境下，我们通常所说的“报文”是 ISO 的 7 层模型中第 4 层，即“传输层”的概念；而具体在网络中发送和接收的则是第 3 层，即“网络层”的数据单位，称为 packet（“报文分组”，或“包”）。根据长度，一个报文可以就是一个包，也可以在发送端被分解成若干个包，而在接收端予以组装复原。报文与包之间的这种区分主要是为克服网络中物理介质的不可靠性，以及从性能方面考虑而来的。在 Unix 域的条件下，由于并不涉及网络介质，所以一个报文就是一个包。每一个包都要占用一个 sk_buff 数据结构，所以 receive_queue 队列中的每个 sk_buff 数据结构就载运着一个到达的包，而 write_queue 队列中则为待发送的包。这也是一种比较复杂的数据结构，我们将结合报文的发送和接收加以介绍。

此外，在 sock 结构中还有一个特殊的 sk_buff 结构队列，那是专为网络环境而设置的，我们在这里并不关心。在网络环境下，常常要为包的发送设置一些定时器。例如，在“有连接”模式中如果发送了一个包以后在一定时间里得不到对方的确认就要重发。所以在 sock 结构中还有一个定时器队列 timer。所有这些队列的头部都要加以初始化（见 1125~1130 行）。

函数 sock_init_data( )中其余的代码比较直截了当，我们就不多解释了。只说明一下，sock 结构中的 rcvbuf 和 sndbuf 分别为接收和发送缓冲区的大小，默认值均为 64K 字节。还有，sock 结构中的 state_change、data_ready 等都是函数指针，分别设置成指向 sock_def_wakeup( )、sock_def_readable( )等函数。

回到 unix_create1( )的代码中，继续完成 sock 结构的初始化。注意代码中的 sock 是一个 socket 结构指针，sk 才是 sock 结构指针。

这里的 sk->write_space 也是一个函数指针，设置成指向函数 unix_write_space( )。在 sock 数据结构中还有一个非常重要的成分 protinfo，这是个 union，要根据具体网域而赋予不同的解释。对 Unix 域来说，这个 union 被当作一个 unix_opt 数据结构，名为 af_unix，这种数据结构是在 include/net/sock.h 中定义的：

```
106    /* The AF_UNIX specific socket options */
107    struct unix_opt {
108        struct unix_address *addr;
109        struct dentry *      dentry;
110        struct vfsmount *    mnt;
111        struct semaphore     readsem;
112        struct sock *        other;
113        struct sock **       list;
114        struct sock *        gc_tree;
115        atomic_t             inflight;
116        rwlock_t             lock;
117        wait_queue_head_t    peer_wait;
118    };
```

这里的 addr 指向一个 unix_address 结构。将一个插口 bind 到某一个地址时，这个地址就保存在这里，其类型定义在 include/net/af_unix.h 中：

```
20     struct unix_address
21     {
22         atomic_t    refcnt;
23         int         len;
24         unsigned    hash;
25         struct sockaddr_un name[0];
26     };
```

具体的地址在 sockaddr_un 结构中，这种数据结构的定义在 include/linux/un.h 中：

```
4      #define UNIX_PATH_MAX    108
5
```

```
6     struct sockaddr_un {
7         sa_family_t sun_family; /* AF_UNIX */
8         char sun_path[UNIX_PATH_MAX];   /* pathname */
9     };
```

可见，Unix 域中的插口地址即为一个文件（节点）的路径名。注意在 unix_address 结构中的 name[ ] 数组大小为 0，所以 unix_address 结构的大小并不包括 sockaddr_un 结构的空间，在分配空间时要额外加上。那么，为什么要这样呢？这是因为插口的地址可长可短（最长为 108 个字节），如果固定按最大长度分配空间就太浪费了。

除 addr 以外，af_unix 结构中的 readsem 是个内核信号量，代码中的 init_MUTEX( )将这个内核信号量的值初始化成 1，用来保证某些操作对于资源的独占性。

对 sock 结构初始化的最后一步是 unix_insert_socket( )，这是个 inline 函数，其定义为：

[sys_socketcall( ) > sys_socket( ) > sock_create( ) > unix_create( ) > unix_create1( ) > unix_insert_socket ( )]

```
241     {static __inline__ void unix_insert_socket(unix_socket **list, unix_socket *sk)
242     {
243         write_lock(&unix_table_lock);
244         __unix_insert_socket(list, sk);
245         write_unlock(&unix_table_lock);
246     }
```

函数__unix_insert_socket( )则将一个 sock 数据结构插入到一个 unix_socket 结构队列中，并将该 sock 结构的使用计数设置成 1。内核中设置了一个杂凑表 unix_socket_table[ ]，其大小为 UNIX_HASK_SIZE+1，即 257。定义见 af_unix.c：

```
118     unix_socket *unix_socket_table[UNIX_HASH_SIZE+1];
```

这个数组中的每一项都是一个 unix_socket 结构的单链队列。在 include/net/af_unix.h 中有对 unix_socket 的定义：

```
6     typedef struct sock unix_socket;
```

所以 unix_socket 结构就是 sock 结构。每个 sock 结构根据其插口地址的杂凑值挂入这个杂凑表中的某个队列。这样，当接收到一个无连接模式的报文时，或接收到一个连接请求时，就可以根据其目标地址迅速地找到其目标插口。但是，在插口创建之初尚无插口地址，所以暂时把它链入到杂凑表中的最后一个队列，即 unix_socket_table[UNIX_HASH_SIZE]。由于地址的杂凑值都在[0，UNIX_HASH_SIZE－1]范围内，所以不会引起混淆。代码中的 unix_sockets_unbound 在 af_unix.c 中定义为：

```
120     #define unix_sockets_unbound (unix_socket_table[UNIX_HASH_SIZE])
```

当然，它在这个队列中只是暂时栖身而已。到用户程序调用 bind( )的时候，就会根据地址的杂凑值把插口的 sock 结构转移到杂凑表中的另一个队列中去。

前面讲过，socket 结构与 sock 结构实际上是同一事物的两个侧面。不妨说，socket 结构是面向进程和系统调用界面的侧面，而 sock 结构则是面向底层驱动程序的侧面。可是，为什么不干脆把两个数据结构合并成一个呢？如前所述，socket 结构是 inode 结构中的一部分，即把 inode 结构内部的一个 union 用作 socket 结构。由于插口操作的特殊性，这个数据结构中需要有大量的结构成分。可是，如果把这些结构成分全都放在 socket 结构中，则 inode 结构中的这个 union 就会变得很大，从而 inode 结构也会变得很大，而对于其他文件系统这个 union 是不需要那么庞大的。所以，那样会使 inode 结构变得过分庞大而造成浪费，系统中使用 inode 结构的数量当然要远远超过使用 socket 结构的数量。解决的办法就是把插口所需的这些结构成分拆成两部分，把与文件系统关系比较密切的那一部分放在 socket 结构中，另一部分，即与通信关系比较密切的那一部分，则单独成为一个数据结构，那就是 sock 结构。由于这两部分数据在逻辑上本来就是一体的，所以要通过指针互相指向对方，形成一对一的关系。

完成了 socket 结构和 sock 结构的分配以及初始化，sock_create( )就完成了任务，我们回到 sys_socket( )中继续往下看。下一步是 sock_map_fd( )，代码见 net/socket.c：

[sys_socketcall( ) > sys_socket( ) > sock_map_fd( )]

```
312    /*
313     *  Obtains the first available file descriptor and sets it up for use.
314     *
315     *  This functions creates file structure and maps it to fd space
316     *  of current process. On success it returns file descriptor
317     *  and file struct implicitly stored in sock->file.
318     *  Note that another thread may close file descriptor before we return
319     *  from this function. We use the fact that now we do not refer
320     *  to socket after mapping. If one day we will need it, this
321     *  function will inincrement ref. count on file by 1.
322     *
323     *  In any case returned fd MAY BE not valid!
324     *  This race condition is inavoidable
325     *  with shared fd spaces, we cannot solve is inside kernel,
326     *  but we take care of internal coherence yet.
327     */
328
329    static int sock_map_fd(struct socket *sock)
330    {
331        int fd;
332        struct qstr this;
333        char name[32];
334
335        /*
336         *  Find a file descriptor suitable for return to the user.
337         */
338
339        fd = get unused fd( );
340        if (fd >= 0) {
341            struct file *file = get_empty_filp( );
```

```
342
343             if (!file) {
344                 put_unused_fd(fd);
345                 fd = -ENFILE;
346                 goto out;
347             }
348
349             sprintf(name, "[%lu]", sock->inode->i_ino);
350             this.name = name;
351             this.len = strlen(name);
352             this.hash = sock->inode->i_ino;
353
354             file->f_dentry = d_alloc(sock_mnt->mnt_sb->s_root, &this);
355             if (!file->f_dentry) {
356                 put_filp(file);
357                 put_unused_fd(fd);
358                 fd = -ENOMEM;
359                 goto out;
360             }
361             file->f_dentry->d_op = &sockfs_dentry_operations;
362             d_add(file->f_dentry, sock->inode);
363             file->f_vfsmnt = mntget(sock_mnt);
364
365             sock->file = file;
366             file->f_op = sock->inode->i_fop = &socket_file_ops;
367             file->f_mode = 3;
368             file->f_flags = O_RDWR;
369             file->f_pos = 0;
370             fd_install(fd, file);
371         }
372
373     out:
374         return fd;
375     }
```

前面讲过，从进程的角度来看，一个插口就是一个特殊的已打开文件，这是在设计“插口”这种机制时就设计好了的。现在 socket 结构（以及 sock 结构）已经分配好并进行了初始化，接着要做的就是将 socket 结构，实际上是其宿主 inode 结构，与文件系统的一套机制挂上钩了。首先要分配一个空闲的打开文件号以及 file 结构；还要在文件系统的目录树中分配一个 dentry 数据结构，使其指向已经分配的 inode 数据结构；并且使 file 结构中的指针 f_dentry 指向该 dentry 结构。最后，还会根据所建立的 dentry 结构和 inode 结构在磁盘上建立起相应的目录项和索引节点，不过那是后话。从代码中还可看出，代表着插口的节点名就是其索引节点号（349 行）。

代码中 354 行的 d_alloc( )为插口分配一个目录项，并将其挂在特殊文件系统 sockfs 的根目录下。然后，362 行的 d_add( )将插口的 dentry 结构与 inode 结构互相挂上钩。这里把 dentry 结构中的指针 d_op 设置成指向 sockfs_dentry_operations，这个数据结构通过函数指针提供了与文件路径有关的操作，定义于 net/socket.c 中：

```
308     static struct dentry_operations sockfs_dentry_operations = {
309         d_delete:    sockfs_delete_dentry,
310     };
```

可见，对于插口，除 vfs 层上的操作以外惟一与此有关的操作就是删除目录项。

最后，将 file 结构中的 f_op 指针和 inode 结构中的 i_fop 指针都设置成指向用于插口的文件操作跳转结构 socket_fs_ops，并根据分配得到的打开文件号 fd 将该 file 结构的地址填入本进程的打开文件结构数组中相应的位置上。数据结构 socket_file_ops 的定义为（socket.c）：

```
114     static struct file_operations socket_file_ops = {
115         llseek:      sock_lseek,
116         read:        sock_read,
117         write:       sock_write,
118         poll:        sock_poll,
119         ioctl:       sock_ioctl,
120         mmap:        sock_mmap,
121         open:        sock_no_open,    /* special open code to disallow open via /proc */
122         release:     sock_close,
123         fasync:      sock_fasync,
124         readv:       sock_readv,
125         writev:      sock_writev
126     };
```

这样，当通过 sys_socketcall( )对一个插口进行某种操作时（如 recv( )等），就会通过前述 socket 结构中的 unix_stream_ops 或 unix_dgram_ops 结构跳转到具体的驱动程序中；而通过普通的文件操作界面（如 read( )等）对插口操作时，则通过这个 socket_fs_ops 结构决定跳转。

完成了 sock_mapfd( )以后，整个创建插口的过程就完成了。总结以上所述，我们可以粗略地画出内核中与插口有关的主要数据结构之间的联系图（图 7.2）。

图中有两个入口。一个是进程的 task_struct 数据结构，根据打开文件号可以找到相应的 inode 结构；另一个是从杂凑表 unix_socket_table 开始找到相应的 sock 结构。当接收到一个 packet 时，在 packet 的头部必然直接或间接地包含着关于目标插口地址的信息。根据这个地址，加以杂凑运算后便可以确定 unix_socket_table 表中的一个队列，然后扫描该队列加以比对，就可以找到属于该目标地址的 sock 数据结构，而找到了 sock 结构也就找到了 socket 结构以及 inode 结构。

最后还要说明，除 SYS_SOCKET 以后，还有个 SYS_SOCKETPAIR 操作也用来创建插口。SYS_SOCKETPAIR 所建立的是一对（而不是一个）互相已经连接好的插口，概念上与“管道”相似。使用上也很相似（例如都要与 fork( )结合使用）。事实上，在有些 Unix 版本中甚至用“插口对”来实现“管道”。由于“插口对”只能在同一台计算机上使用，就没有必要再根据报文的目标地址通过杂凑表找到目标插口的 sock 结构，而可以把这一步“短路”过去了。为了这个目的，在 sock 数据结构中设置了一个指针 pair。在创建一个“插口对”时就让两个 sock 数据结构都通过这个指针互相指向对方。SYS_SOCKETPAIR 操作是由 sys_socketpair( )实现的，有兴趣的读者可以自行阅读其代码。

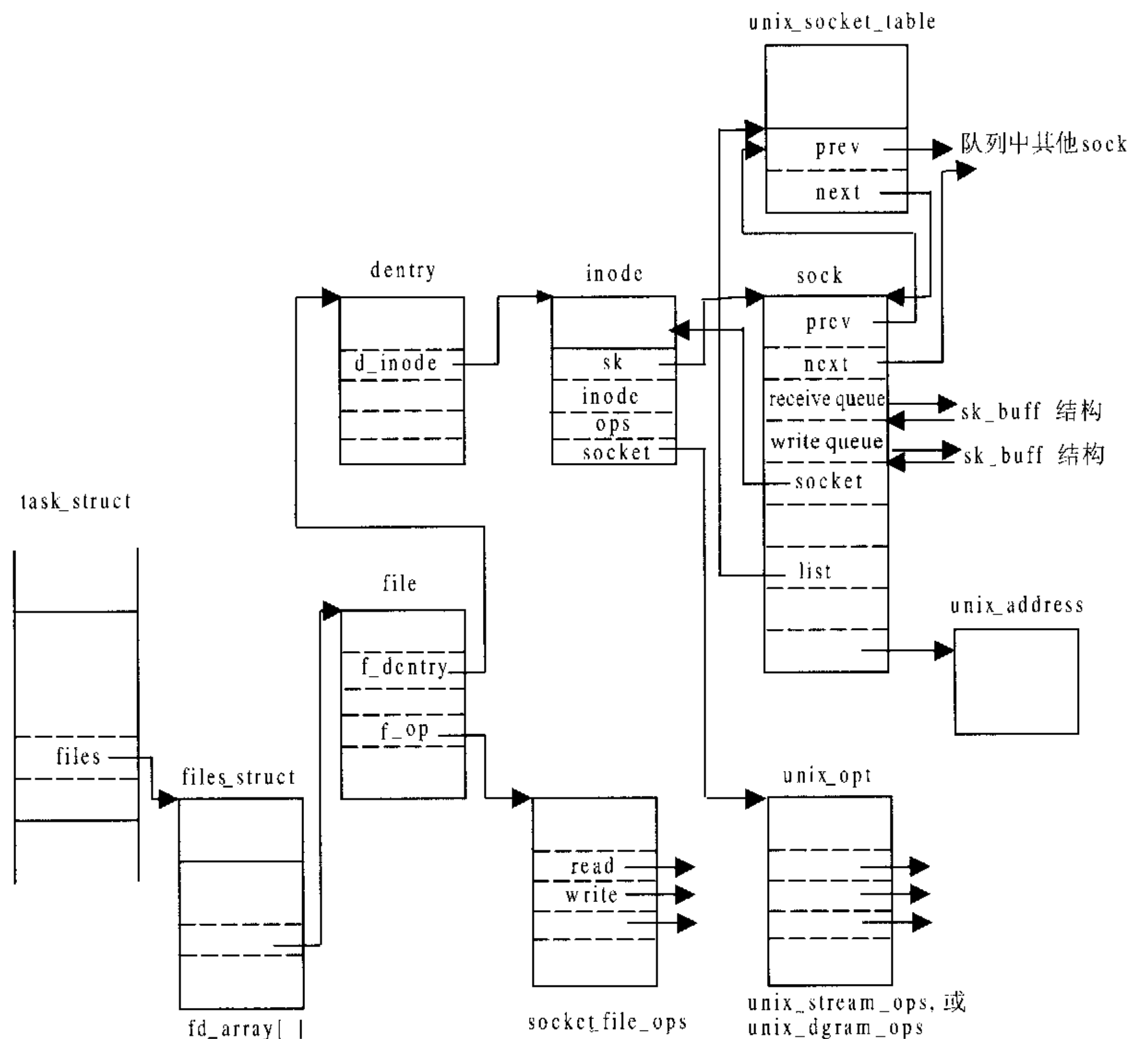


图 7.2　与 socket 相关的主要数据结构之间的联系图

7.3　函数 sys_bind( )——指定插口地址

操作 SYS_BIND 为已创建的插口指定一个地址，是由 sys_bind( )实现的（socket.c）:

[sys_socketcall( ) > sys_bind( )]

```
977    /*
978     *  Bind a name to a socket. Nothing much to do here since it's
979     *  the protocol's responsibility to handle the local address.
980     *
981     *  We move the socket address to kernel space before we call
982     *  the protocol layer (having also checked the address is ok).
```

```
983      */
984
985     asmlinkage long sys_bind(int fd, struct sockaddr *umyaddr, int addrlen)
986     {
987         struct socket *sock;
988         char address[MAX_SOCK_ADDR];
989         int err;
990
991         if((sock = sockfd_lookup(fd,&err))!=NULL)
992         {
993             if((err=move_addr_to_kernel(umyaddr,addrlen,address))>=0)
994             err = sock->ops->bind(sock, (struct sockaddr *)address, addrlen);
995             sockfd_put(sock);
996         }
997         return err;
998     }
```

首先要根据插口的打开文件号找到它的 socket 数据结构，sockfd_lookup( )的代码很简单，读者可以结合前面的图 7.2 自行阅读（socket.c）。

[sys_socketcall( ) > sys_bind( ) > sockfd_lookup( )]

```
382     /**
383      *  sockfd_lookup   -   Go from a file number to its socket slot
384      *  @fd: file handle
385      *  @err: pointer to an error code return
386      *
387      *  The file handle passed in is locked and the socket it is bound
388      *  too is returned. If an error occurs the err pointer is overwritten
389      *  with a negative errno code and NULL is returned. The function checks
390      *  for both invalid handles and passing a handle which is not a socket.
391      *
392      *  On a success the socket object pointer is returned.
393      */
394
395     struct socket *sockfd_lookup(int fd, int *err)
396     {
397         struct file *file;
398         struct inode *inode;
399         struct socket *sock;
400
401         if (!(file = fget(fd)))
402         {
403             *err = -EBADF;
404             return NULL;
405         }
406
```

```
407         inode = file->f_dentry->d_inode;
408         if (!inode->i_sock || !(sock = socki_lookup(inode)))
409         {
410             *err = -ENOTSOCK;
411             fput(file);
412             return NULL;
413         }
414
415         if (sock->file != file) {
416             printk(KERN_ERR "socki_lookup: socket file changed!\n");
417             sock->file = file;
418         }
419         return sock;
420     }
```

找到了插口的 socket 结构以后，就可以根据该结构中的指针 ops 找到相应的驱动程序跳转结构 unix_stream_ops 或 unix_dgram_ops。两个跳转结构中的指针 bind 都指向 unix_bind( )，所以 Unix 域的 bind 操作是由 unix_bind( )完成的。

不过，先要把参数 umyaddr 所指的数据结构从用户空间复制过来，这个数据结构含有为插口指定的地址。前面在 sys_socketcall( )复制过来的只是指向这个数据结构的指针，而不是数据结构本身。这里（988 行）为插口的地址准备下了一个缓冲区 address[ ]，其大小为 MAX_SOCK_ADDR，定义为 128。具体的复制是由 move_addr_to_kernel( )完成的，复制的长度则取决于参数 addrlen。

参数 umyaddr 是个 sockaddr 结构指针，我们还没有看过这种数据结构的定义，现在需要看一下了，这是在文件 include/linux/socket.h 中给出的:

```
11    typedef unsigned short  sa_family_t;
12
13    /*
14     *  1003.1g requires sa_family_t and that sa_data is char.
15     */
16
17    struct sockaddr {
18        sa_family_t sa_family;  /* address family, AF_xxx   */
19        char        sa_data[14];    /* 14 bytes of protocol address */
20    };
```

虽然 sockaddr 数据结构有确定的大小，其中的 sa_data[ ]是一个 14 字节的数组，但是插口地址的实际长度却并不是固定的，所以由另一个参数 addrlen 说明其包括 sa_family 在内的长度。读者也许感到奇怪：sa_data[ ]中不是一个字符串吗？只要用 strlen( )取字符串长度就行了，何必这么麻烦？其实不然，就 Unix 域来说，这通常是个字符串，可是在其他网域中就未必了，例如在 internet 网域中就可以是 4 个字节的 IP 地址加上传输层的端口号，其中有些字节完全可能是 0（后面读者会看到，即使在 Unix 域中也还有所谓“抽象地址”，是以“\0”开头的）。另一方面，Unix 域的插口地址实际上是个文件路径名，14 个字节显然是不够的。所以，用于插口地址的数据结构大小因网域而异，可是结构中的第一个成分总是 sa_family。这么一考虑，读者就可以明白当初设计者（从 BSD Unix 时代开始）的用意了。

在极端的情况下，地址的长度 addrlen 可以只包含了数据类型 sa_family 的长度，也就是一个短整数的长度，而 sa_data[ ]中则连一个字节也没有。设计者赋予这样的地址一个特殊的含义，就是让系统给自动分配（生成）一个地址。对于 Unix 域插口地址，实际使用的是 sockaddr_un 结构，与 sockaddr 结构的区别在于它的字符数组大小为 UNIX_PATH_MAX，实际上是 108。注意 address[ ]的大小为 MAX_SOCK_ADDR，即 128，已经考虑到了对所有网域插口地址的需要。由于在 address[ ]中有足够大的空间，将 sockaddr_un 结构复制过来不会造成问题。常数 MAX_SOCK_ADDR 的定义和 move_addr_to_kernel( )的代码都在 net/socket.c 中：

```
194   #define MAX_SOCK_ADDR   128     /* 108 for Unix domain -
195                                   16 for IP, 16 for IPX,
196                                   24 for IPv6,
197                                   about 80 for AX.25
198                                   must be at least one bigger than
199                                   the AF_UNIX size (see net/unix/af_unix.c
200                                   :unix_mkname( )).
201                                 */
202
203   /**
204    *  move_addr_to_kernel -   copy a socket address into kernel space
205    *  @uaddr: Address in user space
206    *  @kaddr: Address in kernel space
207    *  @ulen: Length in user space
208    *
209    *  The address is copied into kernel space. If the provided address is
210    *  too long an error code of -EINVAL is returned. If the copy gives
211    *  invalid addresses -EFAULT is returned. On a success 0 is returned.
212    */
213
214   int move_addr_to_kernel(void *uaddr, int ulen, void *kaddr)
215   {
216       if(ulen<0||ulen>MAX_SOCK_ADDR)
217           return -EINVAL;
218       if(ulen==0)
219           return 0;
220       if(copy_from_user(kaddr,uaddr,ulen))
221           return -EFAULT;
222       return 0;
223   }
```

其主体是 copy_from_user( )，读者对此已不陌生了。

如前所述，Unix 域的 bind 操作是由 unix_bind( )完成的，其代码在 net/unix/af_unix.c 中：

[sys_socketcall( ) > sys_bind( ) > unix_bind( )]

```
636   static int unix_bind(struct socket *sock, struct sockaddr *uaddr, int addr_len)
637   {
```

```
638        struct sock *sk = sock->sk;
639        struct sockaddr_un *sunaddr=(struct sockaddr_un *)uaddr;
640        struct dentry * dentry = NULL;
641        struct nameidata nd;
642        int err;
643        unsigned hash;
644        struct unix_address *addr;
645        unix_socket **list;
646
647        err = -EINVAL;
648        if (sunaddr->sun_family != AF_UNIX)
649            goto out;
650
651        if (addr_len==sizeof(short)) {
652            err = unix_autobind(sock);
653            goto out;
654        }
655
656        err = unix_mkname(sunaddr, addr_len, &hash);
657        if (err < 0)
658            goto out;
659        addr_len = err;
660
661        down(&sk->protinfo.af_unix.readsem);
662
663        err = -EINVAL;
664        if (sk->protinfo.af_unix.addr)
665            goto out_up;
666
667        err = -ENOMEM;
668        addr = kmalloc(sizeof(*addr)+addr_len, GFP_KERNEL);
669        if (!addr)
670            goto out_up;
671
672        memcpy(addr->name, sunaddr, addr_len);
673        addr->len = addr_len;
674        addr->hash = hash^sk->type;
675        atomic_set(&addr->refcnt, 1);
676
677        if (sunaddr->sun_path[0]) {
678            err = 0;
679            /*
680             * Get the parent directory, calculate the hash for last
681             * component.
682             */
683            if (path_init(sunaddr->sun_path, LOOKUP_PARENT, &nd))
684                err = path_walk(sunaddr->sun_path, &nd);
685            if (err)
```

```
686                 goto out_mknod_parent;
687             /*
688              * Yucky last component or no last component at all?
689              * (foo/., foo/.., /////)
690              */
691             err = -EEXIST;
692             if (nd.last_type != LAST_NORM)
693                 goto out_mknod;
694             /*
695              * Lock the directory.
696              */
697             down(&nd.dentry->d_inode->i_sem);
698             /*
699              * Do the final lookup.
700              */
701             dentry = lookup_hash(&nd.last, nd.dentry);
702             err = PTR_ERR(dentry);
703             if (IS_ERR(dentry))
704                 goto out_mknod_unlock;
705             err = -ENOENT;
706             /*
707              * Special case - lookup gave negative, but... we had foo/bar/
708              * From the vfs_mknod( ) POV we just have a negative dentry -
709              * all is fine. Let's be bastards - you had / on the end, you've
710              * been asking for (non-existent) directory. -ENOENT for you.
711              */
712             if (nd.last.name[nd.last.len] && !dentry->d_inode)
713                 goto out_mknod_dput;
714             /*
715              * All right, let's create it.
716              */
717             err = vfs_mknod(nd.dentry->d_inode, dentry,
718                 S_IFSOCK|sock->inode->i_mode, 0);
719             if (err)
720                 goto out_mknod_dput;
721             up(&nd.dentry->d_inode->i_sem);
722             dput(nd.dentry);
723             nd.dentry = dentry;
724
725             addr->hash = UNIX_HASH_SIZE;
726         }
727
728         write_lock(&unix_table_lock);
729
730         if (!sunaddr->sun_path[0]) {
731             err = -EADDRINUSE;
732             if (__unix_find_socket_byname(sunaddr, addr_len,
733                               sk->type, hash)) {
```

```
734                 unix_release_addr(addr);
735                 goto out_unlock;
736             }
737 
738             list = &unix_socket_table[addr->hash];
739         } else {
740         list = &unix_socket_table[dentry->d_inode->i_ino & (UNIX_HASH_SIZE-1)];
741             sk->protinfo.af_unix.dentry = nd.dentry;
742             sk->protinfo.af_unix.mnt = nd.mnt;
743         }
744 
745         err = 0;
746         __unix_remove_socket(sk);
747         sk->protinfo.af_unix.addr = addr;
748         __unix_insert_socket(list, sk);
749 
750     out_unlock:
751         write_unlock(&unix_table_lock);
752     out_up:
753         up(&sk->protinfo.af_unix.readsem);
754     out:
755         return err;
756 
757     out_mknod_dput:
758         dput(dentry);
759     out_mknod_unlock:
760         up(&nd.dentry->d_inode->i_sem);
761     out_mknod:
762         path_release(&nd);
763     out_mknod_parent:
764         if (err==-EEXIST)
765             err=-EADDRINUSE;
766         unix_release_addr(addr);
767         goto out_up;
768     }
```

对于Unix域的插口地址，代码中先把它的sockaddr结构当成一个sockaddr_un结构（638行），后者的字符数组大小为UNIX_PATH_MAX，即108。此时实际的插口地址已经在sys_bind( )中的局部量address[ ]中，其大小为128，所以不会造成问题。

如前所述，如果参数addrlen指示插口地址的大小为2，即仅为数据类型sa_family_t的大小，则表示要求由系统给自动分配（生成）一个地址。这就是652行中对unix_autobind( )的调用。这个函数的代码也在af_unix.c中，建议读者在按正常路线读完unix_bind( )的代码以后自己再看一下unix_autobind( )。

虽然move_addr_to_kernel( )对插口地址的长度作了检查，对其内容却并未触及，所以要进一步通过unix_mkname( )检查和处理。顺便提一句，变量名“sunaddr”中的s表示socket，un表示unix，而与“Sun”电脑公司毫无关系。unix_mkname( )的代码在文件af_unix.c中：

```
170	/*
171	 *  Check unix socket name:
172	 *      - should be not zero length.
173	 *          - if started by not zero, should be NULL terminated (FS object)
174	 *      - if started by zero, it is abstract name.
175	 */
176	
177	static int unix_mkname(struct sockaddr_un * sunaddr, int len, unsigned *hashp)
178	{
179	    if (len <= sizeof(short) || len > sizeof(*sunaddr))
180	        return -EINVAL;
181	    if (!sunaddr || sunaddr->sun_family != AF_UNIX)
182	        return -EINVAL;
183	    if (sunaddr->sun_path[0])
184	    {
185	        /*
186	         *  This may look like an off by one error but it is
187	         *  a bit more subtle. 108 is the longest valid AF_UNIX
188	         *  path for a binding. sun_path[108] doesnt as such
189	         *  exist. However in kernel space we are guaranteed that
190	         *  it is a valid memory location in our kernel
191	         *  address buffer.
192	         */
193	        if (len > sizeof(*sunaddr))
194	            len = sizeof(*sunaddr);
195	        ((char *)sunaddr)[len]=0;
196	        len = strlen(sunaddr->sun_path)+1+sizeof(short);
197	        return len;
198	    }
199	
200	    *hashp = unix_hash_fold(csum_partial((char*)sunaddr, len, 0));
201	    return len;
202	}
```

Unix 域的插口地址有两种类型。一种是常规的路径名字符串，不过不一定以 0 结尾，在长度中也不包括结尾的 0 在内；另一种是以“\0”开头的，称为抽象地址。对于前者，unix_mkname( )将其转换成一个以 0 结尾的字符串，并对其长度作出相应调整（195～196 行）。后者类似于网络地址，unix_mkname( )为之计算出一个杂凑值，并通过参数 hashp 返回这个杂凑值。

对于这两种不同类型地址的使用和处理是不同的。对常规的字符串地址，unix_bind( )根据其路径名为之在文件系统中建立一个“文件”节点。像其他特殊文件一样，这个文件实际上只是一个索引节点，而并没有用于数据的记录块，但是这样一来这个插口在文件系统的目录树中就成为可见的了。以后就可以通过 pathwalk( )根据该插口的地址（路径名）在文件系统中找到其 inode，并用索引节点号代替地址的杂凑值来决定将插口的 sock 数据结构挂入杂凑表中的哪个队列。对于“抽象地址”则并不建立这样的文件，所以使用“抽象地址”的 Unix 域插口就像用于其他网域的插口一样，从文件系统的角度来讲是不可见的，并且使用地址的杂凑值来确定挂入到哪个队列中。

回到 unix_bind( )的代码中。当插口地址为常规的文件路径名时(677～726 行)，首先通过 path_init( )和 path_walk( )根据给定的路径名找到其父节点，即所在的目录节点，并确认目标节点尚不存在。学习过“文件系统”一章的读者对这些代码不应该有困难。

然后，通过 vfs_mknod( )在文件系统的目录树中建立一个文件节点（见 717 行）。这个函数的代码我们在“文件系统”一章中并未触及，所以在这里看一下，它在 fs/namei.c 中：

[sys_socketcall( ) > sys_bind( ) > unix_bind( ) > vfs_mknod( )]

```
1176    int vfs_mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t dev)
1177    {
1178        int error = -EPERM;
1179
1180        mode &= ~current->fs->umask;
1181
1182        down(&dir->i_zombie);
1183        if ((S_ISCHR(mode) || S_ISBLK(mode)) && !capable(CAP_MKNOD))
1184            goto exit_lock;
1185
1186        error = may_create(dir, dentry);
1187        if (error)
1188            goto exit_lock;
1189
1190        error = -EPERM;
1191        if (!dir->i_op || !dir->i_op->mknod)
1192            goto exit_lock;
1193
1194        DQUOT_INIT(dir);
1195        lock_kernel( );
1196        error = dir->i_op->mknod(dir, dentry, mode, dev);
1197        unlock_kernel( );
1198    exit_lock:
1199        up(&dir->i_zombie);
1200        if (!error)
1201            inode_dir_notify(dir, DN_CREATE);
1202        return error;
1203        }
```

同样，读者对这段代码不应感到困难。操作的主体是由具体文件系统通过其 inode_operations 结构中的函数指针提供的。以 Ext2 文件系统为例，假设路径名决定了目标节点落在 Ext2 文件系统中，或者说其所在目录在 Ext2 文件系统中，那就会调用 Ext2 的 mknod 操作 ext2_mknod( )。读者在“文件系统”一章中已经阅读过这个函数的代码。读者应该还记得，在创建插口时已经为之建立了一个 inode 数据结构，而现在 vfs_mknod( )显然又会根据插口的路径名在某一文件系统中创建一个索引节点。创建了索引节点以后，当通过系统调用 open( )试图打开这个文件时，就会在内存中根据该索引节点的内容建立起一个 inode 数据结构。那么，这样一来插口岂不就有了两个 inode 结构？两个 inode 结构又各有什么作用？我们知道，在打开文件时，是由一个函数 init_special_inode( )根据文件的模式对其 inode 结

构进行初始化的，而这里在调用 vfs_mknod( )时把文件的模式设置成 S_IFSOCK，所以我们看一下在 init_special_inode( )中对插口文件的处理。

```
200	void init_special_inode(struct inode *inode, umode_t mode, int rdev)
201	{
202		inode->i_mode = mode;
203		if (S_ISCHR(mode)) {
		. . . . . .
212		else if (S_ISSOCK(mode))
213			inode->i_fop = &bad_sock_fops;
		. . . . . .
216	}
```

这里的宏操作 S_ISSOCK( )检查目标节点的模式是否 S_IFSOCK，如果是就把 inode 结构中的 inode_operations 结构指针 I_fop 设置成指向 bad_sock_fops，这个结构的定义在 fs/devices.c 中：

```
196	static struct file_operations bad_sock_fops = {
197		open:		sock_no_open
198	};

191	static int sock_no_open(struct inode *irrelevant, struct file *dontcare)
192	{
193		return -ENXIO;
194	}
```

显然，这个节点是不允许通过常规的系统调用来打开的。另一方面，插口一经创建就是已打开文件，也不需要通过系统调用 open( )来打开。那么，为什么要建立这个文件呢？这是因为 Unix 域允许以常规的路径名作为插口地址，这样的插口地址便于记忆，也便于通过常规的文件操作来检查一个特定的路径名是否已经在使用中。如果文件系统中已经存在具有相同路径名的文件，则 unix_bind( )会失败而返回出错代码 EADDRINUSE。所以通常在用户程序中要在调用 bind( )之前先调用 unlink( )，将可能已经存在的同名文件先删除。应该指出，插口并不是持续存在的，其寿命绝不会超过创建它的进程。当创建插口的进程 exit( )时，它所创建的插口也会随着已打开文件的关闭而消失。可是，为插口创建的文件（结点）却是持续存在的；即使创建它的进程已经 exit( )，甚至机器已经断电，这文件还是存在于磁盘上，所以必须特地加以删除。再看两个 inode 结构间的关系。我们在前一节中讲过，每个插口的 sock 结构都挂在一个杂凑表中的某个队列里。对于采用文件路径名为地址的插口，杂凑值是根据文件的索引节点号产生的，所以只要得到了文件的索引节点号，就可以找到插口的 sock 数据结构，从而找到插口的 socket 结构及其宿主，即创建插口时建立的 inode 结构。

当然，Unix 域的插口也可以不用路径名作为地址，那就是前述的“抽象地址”，其第一个字节必须是 0。采用抽象地址的插口在文件系统中不存在插口文件。

如前所述，插口的 sock 结构在创建插口时暂时挂在 unix_socket_table 的最后一个队列中。这个队列的下标超出了杂凑值的范围，所以用杂凑值作下标是不会访问到这个队列里来的。现在既然已经为插口指定了地址，就要把它的 sock 结构转移到相应的队列中了。对于常规的路径名地址，由于所创建文件的 i 节点号码是惟一的，所以不会重复，其 i 节点号码的低 8 位被用作杂凑表中的下标（见 740 行）。

从本质上讲，取 i 节点号码的低 8 位也是一种杂凑运算，只不过其杂凑函数特别简单而已。“抽象地址”就不同了。在为一个插口指定一个抽象地址时并不能保证这个地址的惟一性，所以要先根据地址的杂凑值在相应的队列中检查一下这个地址是否已经存在。函数__unix_find_socket_byname( )的代码在文件 af_unix.c 中，很简单：

[sys_socketcall( ) > sys_bind( ) > unix_bind( ) > __unix_find_socket_byname ( )]

```
248    static unix_socket *__unix_find_socket_byname(struct sockaddr_un *sunname,
249                                                   int len, int type, unsigned hash)
250    {
251        unix_socket *s;
252
253        for (s=unix_socket_table[hash^type]; s; s=s->next) {
254            if(s->protinfo.af_unix.addr->len==len &&
255               memcmp(s->protinfo.af_unix.addr->name, sunname, len) == 0)
256                return s;
257        }
258        return NULL;
259    }
```

接着，就是把插口的 sock 结构转移到已经确定的队列中了（见 746 行和 748 行）。

最后，回到 sys_bind( )的代码中，sockfd_put( )的作用是递减对插口的 file 结构的使用计数。这是当初在 sockfd_lookup( )中通过 fget( )递增的，所以递减以后不会达到 0。

7.4　函数 sys_listen( )——设定 server 插口

以前讲过，“有连接”模式的插口天生就是按 client/server 的模式运转的，只有在一个 server 插口和一个 client 插口之间才能建立起连接。那么，怎样来区分这两种不同的插口呢？只要在插口创建以后为其调用了 listen( )，这个插口就成为 server 插口了。凡是 server 插口都不能主动去与别的插口建立连接，而只能被动地通过 accept( )接受来自 client 插口的连接请求。而 client 插口则相反，不能调用 accept( )来接受连接请求，而只能主动地通过 connect( )提出连接请求。

我们来看 sys_listen( )的代码（socket.c）：

[sys_socketcall( ) > sys_listen( )]

```
1003    /*
1004     *  Perform a listen. Basically, we allow the protocol to do anything
1005     *  necessary for a listen, and if that works, we mark the socket as
1006     *  ready for listening.
1007     */
1008
1009    asmlinkage long sys_listen(int fd, int backlog)
1010    {
1011        struct socket *sock;
```

```
1012        int err;
1013
1014        if ((sock = sockfd_lookup(fd, &err)) != NULL) {
1015            if ((unsigned) backlog > SOMAXCONN)
1016                backlog = SOMAXCONN;
1017            err=sock->ops->listen(sock, backlog);
1018            sockfd_put(sock);
1019        }
1020        return err;
1021    }
```

与 sys_bind( )的代码比较一下就可以看出，除对参数 backlog 的处理外，二者基本上是一样的，都是根据打开文件号 fd 找到插口的 socket 数据结构（inode 的一部分），进而通过结构中的指针 ops 找到相应的 proto_ops 数据结构 unix_stream_ops 或 unix_dgram_ops，再通过不同的函数指针执行相应的函数。数据结构 unix_dgram_ops 中的指针 listen 设置为 sock_no_listen( )，说明无连接模式的插口并不支持 listen( )，而 unix_stream_ops 中的指针 listem 则设置为 unix_listen( )，此函数在文件 af_unix.c 中：

[sys_socketcall( ) > sys_listen( ) > unix_listen( )]

```
431     static int unix_listen(struct socket *sock, int backlog)
432     {
433         int err;
434         struct sock *sk = sock->sk;
435
436         err = -EOPNOTSUPP;
437         if (sock->type!=SOCK_STREAM)
438             goto out;           /* Only stream sockets accept */
439         err = -EINVAL;
440         if (!sk->protinfo.af_unix.addr)
441             goto out;           /* No listens on an unbound socket */
442         unix_state_wlock(sk);
443         if (sk->state != TCP_CLOSE && sk->state != TCP_LISTEN)
444             goto out_unlock;
445         if (backlog > sk->max_ack_backlog)
446             wake_up_interruptible_all(&sk->protinfo.af_unix.peer_wait);
447         sk->max_ack_backlog=backlog;
448         sk->state=TCP_LISTEN;
449         /* set credentials so connect can copy them */
450         sk->peercred.pid = current->pid;
451         sk->peercred.uid = current->euid;
452         sk->peercred.gid = current->egid;
453         err = 0;
454
455     out_unlock:
456         unix_state_wunlock(sk);
457     out:
458         return err;
```

```
459     }
```

首先，只有插口的类型为 SOCK_STREAM，即“有连接”模式的插口，并且已经为其 bind( )了插口地址，才允许 listen( )。其次，对于符合这些条件的插口也不是什么时候都可以调用 listen( )的。插口的 sock 结构中有个成分 state，用来实现一种“有限状态机”。只有当这个状态机处于 TCP_CLOSE 或 TCP_LISTEN 这两种状态时才可以对其调用 listen( )。在前面 sock_create( )的代码中，我们看到在创建一个插口时要调用函数 sock_init_data( )对分配的 sock 数据结构进行初始化，在那里将 state 设置成 TCP_CLOSE。状态 TCP_CLOSE 表示插口只是刚刚建立，尚未宣称为 server 插口；而 TCP_LISTEN 则表示该插口已经设置成 server 插口，但尚未建立起连接，并且不是在等待来自 client 一方的连接请求。只有在这两种状态下才允许改变插口的参数（主要是连接请求队列的容量）。这里顺便要提一下，“有连接”模式的插口未必都是采用 TCP 规程的，但 TCP 是一种典型的“有连接”规程，所以 sock 结构中实现的是一个 TCP 状态机的子集。这样，即使插口的具体规程并不是 TCP，从插口的使用来说已经能满足要求了。

至于 listen( )真正要做的事情，其实是相当简单的（见 447~452 行）。除状态本身的变化以外，就是设置（或改变）几个参数，主要就是最大队列长度 max_ack_backlog，还有就是“插口主”的身份，即所谓 credentials，具体就是进程的用户号、组号以及进程号。以后，当 server 方接受一个连接请求时，要将这些信息送回给请求连接的一方，让其知道究竟是谁接受了连接请求。还有，当新的队列容量比原来有所扩大时，还要唤醒可能正在睡眠中等待着将连接请求挂入队列的进程（445~446 行）。

7.5　函数 sys_accept( )——接受连接请求

“有连接”模式的插口一旦通过 listen( )设置成 server 插口以后，就只能被动地通过 accept( )接受来自 client 插口的连接请求。进程对 accept( )的调用是阻塞性的，就是说如果没有连接请求就会进入睡眠等待，直到有连接请求到来，接受了请求以后（或者超过了预定的等待时间）才会返回。所以，在已经有连接请求的情况下是“接受连接请求”，而在尚无连接请求的情况下是“等待连接请求”。在内核中，accept( )是通过 net/socket.c 中的函数 sys_accept( )实现的，其代码在 net/socket.c 中：

[sys_socketcall( ) > sys_accept( )]

```
1022    /*
1023     * For accept, we attempt to create a new socket, set up the link
1024     * with the client, wake up the client, then return the new
1025     * connected fd. We collect the address of the connector in kernel
1026     * space and move it to user at the very end. This is unclean because
1027     * we open the socket then return an error.
1028     *
1029     * 1003.1g adds the ability to recvmsg( ) to query connection pending
1030     * status to recvmsg. We need to add that support in a way thats
1031     * clean when we restucture accept also.
1032     */
1033
1034    asmlinkage long sys_accept(int fd, struct sockaddr *upeer sockaddr,
```

```
                                                    int *upeer_addrlen)
1035    {
1036        struct socket *sock, *newsock;
1037        int err, len;
1038        char address[MAX_SOCK_ADDR];
1039
1040        sock = sockfd_lookup(fd, &err);
1041        if (!sock)
1042            goto out;
1043
1044        err = -EMFILE;
1045        if (!(newsock = sock_alloc( )))
1046            goto out_put;
1047
1048        newsock->type = sock->type;
1049        newsock->ops = sock->ops;
1050
1051        err = sock->ops->accept(sock, newsock, sock->file->f_flags);
1052        if (err < 0)
1053            goto out_release;
1054
1055        if (upeer_sockaddr) {
1056        if(newsock->ops->getname(newsock, (struct sockaddr *)address, &len, 2)<0) {
1057                err = -ECONNABORTED;
1058                goto out_release;
1059            }
1060            err = move_addr_to_user(address, len, upeer_sockaddr, upeer_addrlen);
1061            if (err < 0)
1062                goto out_release;
1063        }
1064
1065        /* File flags are not inherited via accept( ) unlike another OSes. */
1066
1067        if ((err = sock_map_fd(newsock)) < 0)
1068            goto out_release;
1069
1070    out_put:
1071        sockfd_put(sock);
1072    out:
1073        return err;
1074
1075    out_release:
1076        sock_release(newsock);
1077        goto out_put;
1078    }
```

说来也许奇怪，实际上一个插口经 listen( )设置成 server 插口以后永远不会与任何插口建立起连接。

因为一旦接受了一个连接请求之后就会创建出另一个插口，连接就建立在这个新的插口与请求连接的那个插口之间，而原先的 server 插口则并无改变，并且还可再次通过 accept( )接受下一个连接请求，就好像母鸡下蛋一样。就这样"只取蛋，不杀鸡"，server 插口永远保持接受新的连接请求的能力，但是其本身从来不成为某个连接的一端。正因为这样，sys_accept( )返回的是新插口的打开文件号，同时还通过作为参数的指针 upeer_sockaddr 返回对方的插口地址。

代码中先通过 sockfd_lookup( )找到 server 插口的 socket 结构，然后通过 sock_alloc( )分配一个新的 socket 结构，为"下蛋"作好准备；再通过相应 proto_ops 数据结构（对于 Unix 域是 unix_stream_ops 或 unix_dgram_ops）中的函数指针 accept 调用具体的函数。从前面数据结构 unix_dgram_ops 的代码可以看到，"无连接"插口的 accept 函数指针设置为 sock_no_accept( )，可见"无连接"插口并不支持 accept( )。而"有连接"插口的 accept 指针则指向 unix_accept( )，此函数在文件 af_unix.c 中：

[sys_socketcall( ) > sys_accept( ) > unix_accept( )]

```
1038    static int unix_accept(struct socket *sock, struct socket *newsock, int flags)
1039    {
1040        unix_socket *sk = sock->sk;
1041        unix_socket *tsk;
1042        struct sk_buff *skb;
1043        int err;
1044
1045        err = -EOPNOTSUPP;
1046        if (sock->type!=SOCK_STREAM)
1047            goto out;
1048
1049        err = -EINVAL;
1050        if (sk->state!=TCP_LISTEN)
1051            goto out;
1052
1053        /* If socket state is TCP_LISTEN it cannot change (for now...),
1054         * so that no locks are necessary.
1055         */
1056
1057        skb = skb_recv_datagram(sk, 0, flags&O_NONBLOCK, &err);
1058        if (!skb)
1059            goto out;
1060
1061        tsk = skb->sk;
1062        skb_free_datagram(sk, skb);
1063        wake_up_interruptible(&sk->protinfo.af_unix.peer_wait);
1064
1065        /* attach accepted sock to socket */
1066        unix_state_wlock(tsk);
1067        newsock->state = SS_CONNECTED;
1068        sock_graft(tsk, newsock);
1069        unix_state_wunlock(tsk);
1070        return 0;
```

```
1071
1072    out:
1073        return err;
1074    }
```

以前我们在讲到插口的创建时曾经提到，在 sock 结构中有几个重要的队列，其中之一是到达报文的队列 receive_queue。到达的报文（更确切地说是 packet）“载运”在 sk_buff 数据结构中，而 receive_queue 等待队列就是 sk_buff 结构的队列。连接请求也是以报文的形式到达的，不过不是一般的数据报文，而属于控制报文。所以，所谓等待连接请求的到来就是等待这种控制报文的到来。另一方面，虽然这里所说插口都是“有连接”模式的，但那只是对数据（报文）而言，而控制报文实际上都是“无连接”的（否则，靠什么手段来建立最初的连接呢?）所以，这里通过 skb_recv_datagram( )从 receive_queue 队列中接收代表着连接请求的控制报文。注意 datagram（“数据报”）并不表示“数据报文”，而是说以“无连接”模式传递的报文，切不要把这二者混淆了。

函数 skb_recv_datagram( )的代码在 net/core/datagram.c 中：

[sys_socketcall( ) > sys_accept( ) > unix_accept( ) > skb_recv_datagram( )]

```
109    /*
110     *  Get a datagram skbuff, understands the peeking, nonblocking wakeups and possible
111     *  races. This replaces identical code in packet, raw and udp, as well as the IPX
112     *  AX.25 and Appletalk. It also finally fixes the long standing peek and read
113     *  race for datagram sockets. If you alter this routine remember it must be
114     *  re-entrant.
115     *
116     *  This function will lock the socket if a skb is returned, so the caller
117     *  needs to unlock the socket in that case (usually by calling skb free_datagram)
118     *
119     *  * It does not lock socket since today. This function is
120     *  * free of race conditions. This measure should/can improve
121     *  * significantly datagram socket latencies at high loads,
122     *  * when data copying to user space takes lots of time.
123     *  * (BTW I've just killed the last cli( ) in IP/IPv6/core/netlink/packet
124     *  *  8) Great win.)
125     *  *                                   --ANK (980729)
126     *
127     *  The order of the tests when we find no data waiting are specified
128     *  quite explicitly by POSIX 1003.1g, don't change them without having
129     *  the standard around please.
130     */
131
132    struct sk_buff *skb_recv_datagram(struct sock *sk, unsigned flags,
                                int noblock, int *err)
133    {
134        int error;
135        struct sk_buff *skb;
136        long timeo;
```

```
137
138      /* Caller is allowed not to check sk->err before skb_recv_datagram( ) */
139        error = sock_error(sk);
140        if (error)
141            goto no_packet;
142
143        timeo = sock_rcvtimeo(sk, noblock);
144
145        do {
146         /* Again only user level code calls this function, so nothing interrupt level
147               will suddenly eat the receive_queue.
148
149               Look at current nfs client by the way...
150               However, this function was corrent in any case. 8)
151             */
152            if (flags & MSG_PEEK)
153            {
154                unsigned long cpu_flags;
155
156                spin_lock_irqsave(&sk->receive_queue.lock, cpu_flags);
157                skb = skb_peek(&sk->receive_queue);
158                if(skb!=NULL)
159                    atomic_inc(&skb->users);
160                spin_unlock_irqrestore(&sk->receive_queue.lock, cpu_flags);
161            } else
162                skb = skb_dequeue(&sk->receive_queue);
163
164            if (skb)
165                return skb;
166
167            /* User doesn't want to wait */
168            error = -EAGAIN;
169            if (!timeo)
170                goto no_packet;
171
172        } while (wait_for_packet(sk, err, &timeo) == 0);
173
174        return NULL;
175
176    no_packet:
177        *err = error;
178        return NULL;
179    }
```

其实我们应该在讲述“无连接”模式的报文接收时再来介绍这个函数，但是既然在这儿先碰上了，也就只好把它提前了。

首先是检查 sock 结构中的出错代码 err，看看从上一次调用这个函数以后至今是否发生了出错，同

时将其清 0。这里的 sock_error( )是个 inline 函数，其定义在 include/net/sock.h 中：

[sys_socketcall( ) > sys_accept( ) > unix_accept( ) > skb_recv_datagram( ) > sock_error( )]

```
1197    /*
1198     *  Recover an error report and clear atomically
1199     */
1200
1201    static inline int sock_error(struct sock *sk)
1202    {
1203        int err=xchg(&sk->err, 0);
1204        return -err;
1205    }
```

如果没有出错的话，就可以试图从队列中接收（脱链）一个报文（即 sk_buff 数据结构）了。这里的标志位 MSG_PEEK 是为 recv( )、recvfrom( )等库函数设置的。在调用那些库函数时可以把参数 flags 中的 MSG_PEEK 置成 1，表示只是看一下队列中是否有报文可接收，但是并不真的接收。除此之外，还有个在文件操作中常用的标志位 O_NONBLOCK，表示如果有报文就接收，但若没有也得马上返回，而不是睡眠等待，这个标志位也可用于 accept( )。进一步，还可以为等待连接的操作设定一个时间限制，这就是参数 noblock 的作用。

大家都知道，队列操作是绝不容许打扰的（这种打扰可能来自其他进程，也有可能来自中断服务程序），所以在 skb_peek( )之前要加锁。至于 skb_dequeue( )，则已经把这一点考虑进去了（见 include/linux/skbuff.h）：

[sys_socketcall( ) > sys_accept( ) > unix_accept( ) > skb_recv_datagram( ) > skb_dequeue( )]

```
513    /**
514     *  skb_dequeue - remove from the head of the queue
515     *  @list: list to dequeue from
516     *
517     *  Remove the head of the list. The list lock is taken so the function
518     *  may be used safely with other locking list functions. The head item is
519     *  returned or %NULL if the list is empty.
520     */
521
522    static inline struct sk_buff *skb_dequeue(struct sk_buff_head *list)
523    {
524        long flags;
525        struct sk_buff *result;
526
527        spin_lock_irqsave(&list->lock, flags);
528        result = __skb_dequeue(list);
529        spin_unlock_irqrestore(&list->lock, flags);
530        return result;
531    }
```

显然，其主体是__skb_dequeue( ):

[sys_socketcall( ) > sys_accept( ) > unix_accept( ) > skb_recv_datagram( ) > skb_dequeue( ) > __skb_dequeue]

```
484    /**
485     *  __skb_dequeue - remove from the head of the queue
486     *  @list: list to dequeue from
487     *
488     *  Remove the head of the list. This function does not take any locks
489     *  so must be used with appropriate locks held only. The head item is
490     *  returned or %NULL if the list is empty.
491     */
492
493    static inline struct sk_buff *__skb_dequeue(struct sk_buff_head *list)
494    {
495        struct sk_buff *next, *prev, *result;
496
497        prev = (struct sk_buff *) list;
498        next = prev->next;
499        result = NULL;
500        if (next != prev) {
501            result = next;
502            next = next->next;
503            list->qlen--;
504            next->prev = prev;
505            prev->next = next;
506            result->next = NULL;
507            result->prev = NULL;
508            result->list = NULL;
509        }
510        return result;
511    }
```

如果队列中有报文的话，接收成功，就可以返回了（见 165 行）。要是没有呢？那就要看在调用 accept( )时的 O_NONBLOCK 标志是否为 1，如果是的话，就马上出错返回（见 170 行），出错代码为 −EAGAIN；否则就通过 wait_for_packet( )睡眠等待。这个函数的代码在 net/core/datagram.c 中：

[sys_socketcall( ) > sys_accept( ) > unix_accept( ) > skb_recv_datagram( ) > wait_for_packet( )]

```
63    static int wait_for_packet(struct sock * sk, int *err, long *timeo_p)
64    {
65        int error;
66
67        DECLARE_WAITQUEUE(wait, current);
68
69        __set_current_state(TASK_INTERRUPTIBLE|TASK_EXCLUSIVE);
```

```
70          add_wait_queue_exclusive(sk->sleep, &wait);
71
72          /* Socket errors? */
73          error = sock_error(sk);
74          if (error)
75              goto out;
76
77          if (!skb_queue_empty(&sk->receive_queue))
78              goto ready;
79
80          /* Socket shut down? */
81          if (sk->shutdown & RCV_SHUTDOWN)
82              goto out;
83
84      /* Sequenced packets can come disconnected. If so we report the problem */
85          error = -ENOTCONN;
86          if(connection_based(sk) && !(sk->state==TCP_ESTABLISHED ||
                                                sk->state==TCP_LISTEN))
87              goto out;
88
89          /* handle signals */
90          if (signal_pending(current))
91              goto interrupted;
92
93          *timeo_p = schedule_timeout(*timeo_p);
94
95      ready:
96          current->state = TASK_RUNNING;
97          remove_wait_queue(sk->sleep, &wait);
98          return 0;
99
100     interrupted:
101         error = sock_intr_errno(*timeo_p);
102     out:
103         current->state = TASK_RUNNING;
104         remove_wait_queue(sk->sleep, &wait);
105         *err = error;
106         return error;
107     }
```

这里 sock 数据结构中的指针 sleep 指向 socket 结构中的队列 wait，是创建插口时在 sock_init_data( ) 中设置好了的。至于数据结构 wait，是由 DECLARE_WAITQUEUE( )定义的一个局部量，其空间在当前进程的系统空间堆栈上，也就是在 wait_for_packet( )的调用框架中，只要不从这个函数返回就一直是有效的。通过 wait_for_packet( )将数据结构 wait 挂入 sock 结构的 sleep 队列，就等于把当前进程挂入了目标插口的（连接请求）等待队列。从这里也可看出，多个进程在同一个插口上等待连接请求是允许的。当前进程通过 schedule_timeout( )进入睡眠以后，就一直要到有下列事件之一发生时才会被唤醒而

从 schedule_timeout( )返回，那就是：①有连接请求到达；②接收到了一个信号；③到达了预定的等待时间。在这三种情况下 wait_for_packet( )的返回值都是 0，所以在 skb_recv_datagram( )的 do-while 循环中都会再执行一次循环体，再作最后一次尝试。此外，wait_for_packet( )还通过指针 timeo_p 返回还剩下的等待时间。

读者也许注意到了，这里的处理与进程在许多系统调用中从睡眠醒来时有所不同。一般，当一个进程从睡眠中醒来时要先检查是否有信号到达，如果有就提前结束本次系统调用，先对信号作出反应。而这里则不同，醒来后还是回到循环体中再试一次，这是为什么呢？实际上，是否要提前结束本次系统调用不能一概而论，要看继续往下执行是否能在一个有限的短时间内完成。在这里的循环体中执行的基本上就是 skb_dequeue( )，那只是举手之劳，下面还会看到在接收到一个报文后的处理也很简单，所以还不如再试一次，哪怕不成功，再来处理信号也还不迟。所以，如果这一次成功了当然最好，那就会在 165 行返回；要是仍不成功，而时间已经到点（timeo 变成了 0），那就会在 170 行结束循环；否则，那就应该是因为接收到信号而被唤醒了（队列中没有报文，时间又未到点），当再次调用 wait_for_packet( )的时候就会在那里的 91 行转到 interrupted 处，从而结束 do-while 循环并返回一个出错代码。至于对信号的处理，则在从系统调用返回时自会按常规进行。

有关连接请求的到达，以及唤醒正在 sys_accept( )中睡眠的进程可以参看后面对 connect( )的介绍，这里我们假定连接请求已经到达了。所以，从 wait_for_packet( )返回以后再试一次就成功了。

回到 unix_accept( )的代码中（1058 行），指针 skb 指向接收到的 sk_buff 数据结构。这种数据结构是在 include/linux/skbuff.h 中定义的，由于要考虑到各种不同的规程，其定义长达将近 100 行。另一方面，我们在本书中所关心的只是 Unix 域，所以就不列出它的定义了。对于 Unix 域的插口，在 sk_buff 数据结构中有个 sock 结构指针 sk，指向一个 sock 数据结构，就是代码中的 tsk。这个 sock 结构是由请求连接的那一方送过来、供接受连接的一方使用的，注意 tsk 与“task”毫无关系，大概是“target sock”的意思。

前面，在 sys_accept( )中（1045 行）已经调用了 sock_alloc( )，分配了一个新的 socket 结构（也就是 inode 结构），那就是 unix_accept( )的参数 newsock。但 sock_alloc( )并不是 sock_create( )，它并不分配与 socket 结构配对的 sock 结构，从这个意义上讲，这个新的插口还不完整。那么什么时候才分配所需的 sock 数据结构呢？这是由 client 一方在 connect( )的过程中分配，并且将其指针通过用作连接请求报文的 sk_buff 结构带过来的，这就是这里的 tsk。现在，就通过 sock_graft( )将 client 一方提供的 tsk 与 server 一方提供的 newsock 挂上钩（1068 行）。

[sys_socketcall( ) > sys_accept( ) > unix_accept( ) > sock_graft ( )]

```
1014    static inline void sock_graft(struct sock *sk, struct socket *parent)
1015    {
1016        write_lock_bh(&sk->callback lock);
1017        sk->sleep = &parent->wait;
1018        parent->sk = sk;
1019        sk->socket = parent;
1020        write_unlock_bh(&sk->callback_lock);
1021    }
```

这样，就完成了主要的连接过程，因为 tsk 与 client 一方的 sock 结构事先已经互相“背靠背”连接好了。同时，新的 socket 结构的状态也直接设置成 SS_CONNECTED。但是请注意，原先的 socket 结

构的状态却并未改变。

读者也许会问，代码中并没有对接收到的报文，即 sk_buff 结构，作什么检查，怎么就能知道它一定是个连接请求报文呢？答案是，只有连接请求才能挂入到 server 插口的 receiver_queue 队列中。数据报文是不能够挂入到这个队列中的，它们只能挂入到建立了连接的新插口的 receive_queue 队列中。另外，也只有在 connect( )的过程中才会唤醒正在等待连接请求的进程。所以，对于 Unix 域的插口来说，是否有“控制报文”这个概念，其实也无关紧要，只是为了与网络环境下的插口机制保持概念上以及实现上的一致性才套用了这些概念。

接受了一个连接请求以后，一方面要释放报文，另一方面要把请求连接的一方唤醒。前面提到过，receive_queue 队列的长度是有限制的，如果队列中的报文数量已经达到了上限，则新到达的报文就因无法“投递”而只好让 client 方进入睡眠等待。现在，server 方接收了一个报文以后，receive_queue 队列的长度下降了，就可以把可能正在睡眠等待的进程唤醒了（1063 行）。这里的 wake_up_interruptible( )是个宏操作，定义于 include/linux/sched.h 中：

```
#define wake_up_interruptible(x)    __wake_up((x),TASK_INTERRUPTIBLE,WQ_FLAG_EXCLUSIVE)
```

调用参数 WQ_FLAG_EXCLUSIVE 表示如果有多个进程在睡眠等待就只唤醒其中一个。

回到 sys_accept( )的代码中。如果用户进程提供了用来返回对方地址的数据结构指针 upeer_sockaddr，就要通过相应 proto_ops 结构中的函数指针 getname 来获取对方的插口地址。在数据结构 unix_stream_ops 中，函数指针 getname 设置为 unix_getname( )，其代码在 af_unix.c 中：

[sys_socketcall( ) > sys_accept( ) > unix_getname( )]

```
1077    static int unix_getname(struct socket *sock, struct sockaddr *uaddr,
                                int *uaddr_len, int peer)
1078    {
1079        struct sock *sk = sock->sk;
1080        struct sockaddr_un *sunaddr=(struct sockaddr_un *)uaddr;
1081        int err = 0;
1082
1083        if (peer) {
1084            sk = unix_peer_get(sk);
1085
1086            err = -ENOTCONN;
1087            if (!sk)
1088                goto out;
1089            err = 0;
1090        } else {
1091            sock_hold(sk);
1092        }
1093
1094        unix_state_rlock(sk);
1095        if (!sk->protinfo.af_unix.addr) {
1096            sunaddr->sun_family = AF_UNIX;
1097            sunaddr->sun_path[0] = 0;
```

```
1098            *uaddr_len = sizeof(short);
1099        } else {
1100            struct unix_address *addr = sk->protinfo.af_unix.addr;
1101
1102            *uaddr_len = addr->len;
1103            memcpy(sunaddr, addr->name, *uaddr_len);
1104        }
1105        unix_state_runlock(sk);
1106        sock_put(sk);
1107    out:
1108        return err;
1109    }
```

这个函数既可以用来获取对方插口的地址，也可以用来获取本插口的地址，具体由参数 peer 决定。在这里由于调用时将参数的值设成 1，所以取的是对方地址。函数 unix_peer_get( )通过 sock 结构中的指针 pair 取得指向对方 sock 结构的指针。如前所述，与 newsock 配对的 sock 结构是由调用 connect( )的一方分配和设置的，当然也包括指针 pair 的设置。所以，在调用了 unix_peer_get( )以后，代码中的指针 sk 就改成指向对方的 sock 结构了。

[sys_socketcall( ) > sys_accept( ) > unix_getname( ) > unix_peer_get ( )]

```
152    static __inline__ unix_socket * unix_peer_get(unix_socket *s)
153    {
154        unix_socket *peer;
155
156        unix_state_rlock(s);
157        peer = unix_peer(s);
158        if (peer)
159            sock_hold(peer);
160        unix_state_runlock(s);
161        return peer;
162    }

140    #define unix_peer(sk) ((sk)->pair)
```

反之，sock_hold( )则只是递增 sock 结构中的使用计数（见 sock.h）：

[sys_socketcall( ) > sys_accept( ) > unix_getname( ) > sock_hold( )]

```
967    /* Grab socket reference count. This operation is valid only
968       when sk is ALREADY grabbed f.e. it is found in hash table
969       or a list and the lookup is made under lock preventing hash table
970       modifications.
971     */
972
973    static inline void sock_hold(struct sock *sk)
```

```
974     {
975         atomic_inc(&sk->refcnt);
976     }
```

前面讲过，server 插口必须有个地址，这样 client 一方才能通过地址来找到 server 插口的数据结构，所以在创建了插口以后一定要调用 bind( )将其“捆绑”到一个地址上。可是，对于 client 一方的插口来说，它只能主动地去寻找某个 server 插口，别的插口是不会来寻找它的。尤其是 Unix 域的插口，所传递的报文都是在同一计算机中而无需穿越网络。所以实际上没有必要让 client 插口也有个地址，当然有也无妨。这样，在 unix_getname( )中就要考虑到两种情况，一种是 sock 结构所属的插口根本就没有地址（1095 行），另一种则是有地址的（1099 行）。

再回到 sys_accept( )的代码中。最后一件事就是为新创建的插口也分配一个打开文件号以及相应的 file 结构，并返回这个打开文件号。函数 sock_map_fd( )的代码读者在 sys_socket( )中已经看到过，这里就不重复了。

7.6 函数 sys_connect( )——请求连接

以前讲过，“有连接”模式的插口与“无连接”模式的插口都可以调用库函数 connect( )，但是意义却不同。内核中的函数 sys_connect( )是二者公用的，只是因“安装”在插口上的 proto_ops 数据结构的不同而通过不同的函数指针转入不同的处理程序。函数 sys_connect( )的代码在文件 net/socket.c 中：

[sys_socketcall( ) > sys_connect( )]

```
1081    /*
1082     *  Attempt to connect to a socket with the server address.  The address
1083     *  is in user space so we verify it is OK and move it to kernel space.
1084     *
1085     *  For 1003.1g we need to add clean support for a bind to AF_UNSPEC to
1086     *  break bindings
1087     *
1088     *  NOTE: 1003.1g draft 6.3 is broken with respect to AX.25/NetROM and
1089     *  other SEQPACKET protocols that take time to connect( ) as it doesn't
1090     *  include the -EINPROGRESS status for such sockets.
1091     */
1092
1093    asmlinkage long sys_connect(int fd, struct sockaddr *uservaddr, int addrlen)
1094    {
1095        struct socket *sock;
1096        char address[MAX_SOCK_ADDR];
1097        int err;
1098
1099        sock = sockfd_lookup(fd, &err);
1100        if (!sock)
1101            goto out;
1102        err = move_addr_to_kernel(uservaddr, addrlen, address);
```

```
1103        if (err < 0)
1104            goto out_put;
1105        err = sock->ops->connect(sock, (struct sockaddr *) address, addrlen,
1106                        sock->file->f_flags);
1107    out_put:
1108        sockfd_put(sock);
1109    out:
1110        return err;
1111    }
```

如前所述，socket 结构中的指针 ops 指向某个 proto_ops 数据结构，对于 unix 域的插口有两种 proto_ops 数据结构，分别用于“有连接” 插口和“无连接”插口。数据结构 unix_stream_ops 中的指针 connect 指向 unix_stream_connect( )，而 unix_dgram_ops 中的这个指针则指向 unix_dgram_connect( )。

先来看“有连接”模式的插口。如前所述，只有 client 插口才可以（并且一定要）通过 connect( ) 向一个 server 插口提出连接请求，在请求被 server 插口接受而建立起连接之前是不能在两个插口之间传递数据报文的。函数 unix_stream_connect( )的代码在 net/unix/af_unix.c 中，我们分段往下看：

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( )]

```
852    static int unix_stream_connect(struct socket *sock, struct sockaddr *uaddr,
853                    int addr_len, int flags)
854    {
855        struct sockaddr_un *sunaddr=(struct sockaddr_un *)uaddr;
856        struct sock *sk = sock->sk;
857        struct sock *newsk = NULL;
858        unix_socket *other = NULL;
859        struct sk_buff *skb = NULL;
860        unsigned hash;
861        int st;
862        int err;
863        long timeo;
864
865        err = unix_mkname(sunaddr, addr_len, &hash);
866        if (err < 0)
867            goto out;
868        addr_len = err;
869
870        if (sock->passcred && !sk->protinfo.af_unix.addr &&
871            (err = unix_autobind(sock)) != 0)
872            goto out;
873
874        timeo = sock_sndtimeo(sk, flags & O_NONBLOCK);
875
876        /* First of all allocate resources.
877           If we will make it after state is locked,
878           we will have to recheck all again in any case.
```

```
879          */
880
881         err = -ENOMEM;
882
883         /* create new sock for complete connection */
884         newsk = unix_create1(NULL);
885         if (newsk == NULL)
886             goto out;
887
888         /* Allocate skb for sending to listening sock */
889         skb = sock_wmalloc(newsk, 1, 0, GFP_KERNEL);
890         if (skb == NULL)
891             goto out;
892
```

这里函数 unix_mkname( )的作用是将目标插口的地址加以某种“规格化”并返回其长度。如果这个地址是一个“抽象地址”，则还要计算出它的杂凑值，其代码我们已经在介绍 sys_socket( )时看到过了。插口的 socket 数据结构中有个标志 passcred，意为“pass credentials”，就是要把自己的“身份”告诉对方。在插口创建之初这个标志为 0，但是可以在用户程序中通过 setsockopt( )将其设置成 1。所谓身份，当然包括地址。所以，如果插口没有地址却又得把身份告诉对方，则必须调用 unix_autobind( )自动生成一个地址（871 行）。

与 accept( )一样，connect( )也是阻塞性的，如果连接请求不能得到 server 方接受就会进入睡眠等待，直到 server 方接受了连接请求（或者超过了预定的等待时间）才会返回。不过也可以把参数 flags 中的 O_NONBLOCK 为标志位，使得在连接请求不能马上得到接受时就立即返回。这里的 sock_sndtimeo( )是个 inline 函数，定义于 include/net/sock.h 中：

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( ) > sock_sndtimeo( )]

```
1249    static inline long sock_sndtimeo(struct sock *sk, int noblock)
1250    {
1251        return noblock ? 0 : sk->sndtimeo;
1252    }
```

然后，通过 unix_create1( )分配一个新的 sock 数据结构，其代码我们已经在介绍 sys_socket( )时看到过。这个新的 sock 数据结构是为 server 一方在调用 accept( )时创建新的插口而准备的。如前所述，一个插口除了有一个 socket 数据结构（实际上是 inode 数据结构的一部分）外，还要有个与之配套使用的 sock 结构。对 Unix 域的“有连接”模式插口而言，这个数据结构是由 client 一方在这里分配的，并将其地址 newsk 通过连接请求报文传递到 server 一方。代表着连接请求的 sk_buff 数据结构则由 sock_wmalloc( )分配并初始化（sock.c）：

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( ) > sock_wmalloc( )]

```
654     /*
655      * Allocate a skb from the socket's send buffer.
```

```
656     */
657     struct sk_buff *sock_wmalloc(struct sock *sk, unsigned long size,
                          int force, int priority)
658     {
659         if (force || atomic_read(&sk->wmem_alloc) < sk->sndbuf) {
660             struct sk_buff * skb = alloc_skb(size, priority);
661             if (skb) {
662                 skb_set_owner_w(skb, sk);
663                 return skb;
664             }
665         }
666         return NULL;
667     }
```

代码中的 sk->wmem_alloc 是 sock 结构中的一个计数器，用来累计为本插口分配的 sk_buff 存储空间，一般不应超过为该插口设置的限额 sk->sndbuf，但是可以通过把参数 force 设成 1 来打破这个限制。参数 size 为随同 sk_buff 结构发送的数据缓冲区的大小，这里在调用时设置成 1，不过实际分配时是以 16 个字节为单位的，所以实际上是 16 个字节。函数 skb_owner_w( )对分配到的 sk_buff 进行一些初始化（sock.h）：

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( ) > sock_wmalloc( ) > skb_owner_w( )]

```
1120    /*
1121     *  Queue a received datagram if it will fit. Stream and sequenced
1122     *  protocols can't normally use this as they need to fit buffers in
1123     *  and play with them.
1124     *
1125     *  Inlined as it's very short and called for pretty much every
1126     *  packet ever received.
1127     */
1128
1129    static inline void skb_set_owner_w(struct sk_buff *skb, struct sock *sk)
1130    {
1131        sock_hold(sk);
1132        skb->sk = sk;
1133        skb->destructor = sock_wfree;
1134        atomic_add(skb->truesize, &sk->wmem_alloc);
1135    }
```

这里使报文缓冲区中的指针 sk 指向为 server 方分配的 sock 数据结构，同时还要把报文缓冲区中的函数指针 destructor 设置成指向 sock_wfree( )，使 server 方将来能按正确的途径释放这个缓冲区。最后，skb->truesize 为包括数据缓冲区在内的实际大小，其初始值是在 alloc_skb( )中设置的。

回到 unix_stream_connect( )中。至此，我们已经分配了一个 sock 结构以及一个空白的 sk_buff 结构。让我们继续往下看（af_unix.c）：

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( )]

```
893     restart:
894         /*  Find listening sock. */
895         other=unix_find_other(sunaddr, addr_len, sk->type, hash, &err);
896         if (!other)
897             goto out;
898 
899         /* Latch state of peer */
900         unix_state_rlock(other);
901 
902         /* Apparently VFS overslept socket death. Retry. */
903         if (other->dead) {
904             unix_state_runlock(other);
905             sock_put(other);
906             goto restart;
907         }
908 
909         err = -ECONNREFUSED;
910         if (other->state != TCP_LISTEN)
911             goto out_unlock;
912 
913         if (skb_queue_len(&other->receive_queue) > other->max_ack_backlog) {
914             err = -EAGAIN;
915             if (!timeo)
916                 goto out_unlock;
917 
918             timeo = unix_wait_for_peer(other, timeo);
919 
920             err = sock_intr_errno(timeo);
921             if (signal_pending(current))
922                 goto out;
923             sock_put(other);
924             goto restart;
925             }
926 
```

函数 unix_find_other( )根据给定的地址找到目标插口的 sock 数据结构，其代码也在 af_unix.c 中：

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( ) > unix_find_other( )]

```
588     static unix_socket *unix_find_other(struct sockaddr_un *sunname, int len,
589                         int type, unsigned hash, int *error)
590     {
591         unix_socket *u;
592         struct nameidata nd;
593         int err = 0;
594 
595         if (sunname->sun_path[0]) {
```

```
596             if (path_init(sunname->sun_path,
597                         LOOKUP_POSITIVE|LOOKUP_FOLLOW, &nd))
598                 err = path_walk(sunname->sun_path, &nd);
599             if (err)
600                 goto fail;
601             err = permission(nd.dentry->d_inode,MAY_WRITE);
602             if (err)
603                 goto put_fail;
604 
605             err = -ECONNREFUSED;
606             if (!S_ISSOCK(nd.dentry->d_inode->i_mode))
607                 goto put_fail;
608             u=unix_find_socket_byinode(nd.dentry->d_inode);
609             if (!u)
610                 goto put_fail;
611 
612             path_release(&nd);
613 
614             err=-EPROTOTYPE;
615             if (u->type != type) {
616                 sock_put(u);
617                 goto fail;
618             }
619         } else {
620             err = -ECONNREFUSED;
621             u=unix_find_socket_byname(sunname, len, type, hash);
622             if (!u)
623                 goto fail;
624         }
625         return u;
626 
627     put_fail:
628         path_release(&nd);
629     fail:
630         *error=err;
631         return NULL;
632     }
```

可见，对于常规的以文件路径名为代表的插口地址，要先通过文件系统的操作 path_init( )和 path_walk( )在文件系统中找到其目录项和索引节点，并在内存中建立起相应的 dentry 结构以及 inode 结构。然后，就使用其索引节点号，通过 unix_find_sock_byinode( )在杂凑表 unix_socket_table 中找到相应的队列和 sock 数据结构。对于“抽象地址”，则使用该地址的杂凑值通过 unix_find_socket_byname( )在 unix_socket_table 中寻找。找到了对方的 sock 结构以后，二者都会通过 sock_hold( )将结构中的访问计数 refcnt 加 1，表示这个结构现在多了一个“用户”。

回到 unix_stream_connect( )的代码中（895 行）。找到了 server 插口的 sock 结构以后，指针 other 指向这个结构。但是，在对这个 sock 结构进行任何操作之前需要确认这个 sock 结构不处于正被撤销的

过程中。虽然对 Unix 域的插口而言，server 和 client 双方都在同一台计算机上，但是在 SMP 多处理器结构下两个进程有可能在两个不同的处理器中运行（但是共用内存）。所以有可能发生这么一种情况，就是当 client 方的进程运行到这里，即从杂凑表的队列中找到了 server 插口的 sock 结构时，恰好在另一个处理器中运行的 server 方的进程已经在关闭、撤销这个插口了。如果不采取有效的措施加以防范，就可能出现严重问题：当 client 方进程在自以为找到了 server 插口的 sock 结构，并且通过 other 指针对其操作（例如将报文挂入它的 receive_queue 队列中）时，事实上 server 方进程已经释放了这个 sock 结构的空间。那么，内核中采取了什么样的措施来防止这种情况的发生呢？

(1) 在 sock 结构中设置了一个访问计数器 refcnt。每当某一进程卷入到这个结构的使用时，或者当所属的插口建立起一个连接时，就要通过 sock_hold( )将这个计数器加 1。前面我们提到过，当 client 方进程从 unix_socket_table 表中的队列里找到 server 插口的 sock 数据结构时，就会通过 sock_hold( )将这个结构中的访问计数 refcnt 加 1，其代码已经在前一节中看到过了。

(2) 相应地，每当一个进程结束了对一个插口的使用时，或者拆除一个连接时，都要通过另一个 inline 函数 sock_put( )将相应 sock 结构中的计数器 refcnt 减 1。只有在这个计数器达到了 0 时，才允许（并且必须）将这个 sock 结构释放：

```
986     /* Ungrab socket and destroy it, if it was the last reference. */
987     static inline void sock_put(struct sock *sk)
988     {
989         if (atomic_dec_and_test(&sk->refcnt))
990             sk_free(sk);
991     }
```

[sock_put( ) > sk_free( )]

```
585     void sk_free(struct sock *sk)
586     {
587     #ifdef CONFIG_FILTER
588         struct sk_filter *filter;
589     #endif
590
591         if (sk->destruct)
592             sk->destruct(sk);
593
594     #ifdef CONFIG_FILTER
595         filter = sk->filter;
596         if (filter) {
597             sk_filter_release(sk, filter);
598             sk->filter = NULL;
599         }
600     #endif
601     aa
602     if (atomic_read(&sk->omem_alloc))
                printk(KERN_DEBUG "sk_free: optmem leakage (%d bytes) detected.\n",
603         atomic_read(&sk->omem_alloc));
604
```

```
605         kmem_cache_free(sk_cachep, sk);
606     }
```

注意 592 行对 destruct 函数的调用，这个函数指针是在分配 sock 数据结构时设置好了的，用来在释放缓冲区之前完成一些必要的附加操作。605 行的 kmem_cache_free( )则将给定的 sock 结构在其 slab 中加以释放。

(3) 关闭、撤销一个插口时，其 sock 结构中的计数 refcnt 有可能还大于 1，表示还有用户，所以不能马上将这个结构释放，而只能将其 refcnt 计数减 1，把释放该结构的责任留给最后将这个计数减到 0 的那个进程。但是，光凭计数 refcnt 不足以说明该插口在逻辑上是否已经撤销，所以在 sock 结构中又设置了一个标志量 dead，表示尽管该 sock 结构中的 refcnt 还不是 0，所以还不能把数据结构最后释放，但实际上插口已经不存在了。

(4) 光是对 sock 结构的使用和释放加以保护还不够，还要防止对 sock 结构的使用（例如报文的到达）和撤销在时间上相重叠。也就是说，这二者在时间上必须加以"串行化"。这样，如果插口的撤销在前，那就让撤销的过程先完成，而对 sock 的使用则就此打住。反之，如果对 sock 结构的使用在前，那就让使用的过程先完成，然后再来撤销，因为在撤销的过程中可能需要对使用的后果（例如链入到 receive_queue 队列中的报文）加以善后处理。为此目的，内核中设置了两对加锁 / 解锁操作，即 unix_state_rlock( )/unix_state_runlock( )和 unix_state_wlock( )/unix_state_wunlock( )。当一个进程要读取 sock 结构中的状态信息（特别是 dead）时，要先调用 unix_state_rlock( )加锁。这样，如果另一个进程正想要改变 sock 结构中的状态信息（例如想要把 dead 变成 1），就要在一个循环中（并不睡眠！）等待解锁后才能继续。反过来，如果一个进程要改变 sock 结构中的状态信息，则要先通过 unix_state_wlock( )加锁。这样，想要读的进程就会在一个循环中等待解锁。这里 unix_state_rlock( )和 unix_state_wlock( )是互锁的，如果 unix_state_rlock( )先成功了，那么调用 unix_state_wlock( )的进程就会在这个函数中循环等待（所以叫 spinlock），反之亦然。

以上这些措施防止了前述问题的发生。在 unix_stream_connect( )的代码中，读者可以看到在检查 other->dead 之前先调用了 unix_state_rlock( )。如果此时 server 方进程正在撤销该插口（及其 sock 数据结构）的过程中，并且赶在前面调用了 unix_state_wlock( )，则当前进程就会在 unix_state_rlock( )中循环等待，直至撤销 sock 数据结构的操作完成。但是此时由于 sock 结构中的 refcnt 计数至少为 2（因为当前进程在从杂凑表 unix_socket_table 的队列中找到这个结构时已经递增了这个计数），所以 server 方进程不会把 sock 结构释放。如果当前进程检测到 other->dead 非 0，就知道这个插口实际上已经不存在了。所以一方面把锁打开，一方面调用 sock_put( )将其 recfnt 计数减 1。如果减 1 以后 refcnt 变成了 0，就担负起释放这个数据结构的责任。然后，就转回标号 restart 处，看看杂凑表的队列中是否还有相同地址的其他 sock 数据结构。当然，通常会因为找不到而使本次连接请求夭折（见 897 行）。

过了这一关，进而要检查目标插口的状态。只有处于 TCP_LISTEN 状态的插口才允许接受连接请求。注意，这与 server 方进程是否已经调用了 accept( )或者正在 accept( )中睡眠等待连接请求无关。

在 server 插口的 sock 结构中已经准备好了容纳连接请求报文的队列。不过，这个队列的长度是有限制的，在 sys_listen( )中已经设置好了这个队列的最大长度。当队列长度超过这个最大长度时，就再不能把新的连接请求链入到队列中了。此时 client 方进程根据预定的等待时间决定是立即出错返回或是睡眠等待。如果决定睡眠等待，就在 918 行调用 unix_wait_for_peer( )进入睡眠（af_unix.c）：

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( ) > unix_wait_for_peer( )]

```
830	static long unix_wait_for_peer(unix_socket *other, long timeo)
831	{
832		int sched;
833		DECLARE_WAITQUEUE(wait, current);
834	
835		__set_current_state(TASK_INTERRUPTIBLE);
836		add_wait_queue_exclusive(&other->protinfo.af_unix.peer_wait, &wait);
837	
838		sched = (!other->dead &&
839			!(other->shutdown&RCV_SHUTDOWN) &&
840			skb_queue_len(&other->receive_queue) > other->max_ack_backlog);
841	
842		unix_state_runlock(other);
843	
844		if (sched)
845			timeo = schedule_timeout(timeo);
846	
847		__set_current_state(TASK_RUNNING);
848		remove_wait_queue(&other->protinfo.af_unix.peer_wait, &wait);
849		return timeo;
850	}
```

与前一节中 server 方在 wait_for_packet( )中睡眠等待相似，这里也是在当前进程的系统空间堆栈上分配一个 wait_queue_t 数据结构，将其挂入目标插口的 peer_wait 队列，然后就通过 schedule_timeout( )进入定时的睡眠，醒来时再从该队列中脱链。在 unix_accept( )的代码中我们看到，当 server 方进程从队列中接收了一个连接请求，从而使队列长度有所下降时，就要唤醒一个（如果有的话）正等待着要将连接请求挂入该队列的进程。当进程被唤醒而从 unix_wait_for_peer( )中返回时，要通过 sock_put( )释放目标插口，再检查是否因为接收到了信号而被唤醒。如果是就出错返回，提前结束本次系统调用；否则就转回到标号 restart 处，从杂凑表中的队列重新开始（见 924 行），因为情况可能已经改变了。

为什么这里与前一节中不同，在接收到了信号而被唤醒时要提前结束本次系统调用呢？我们不妨看看这里等待的是什么。如前所述，这里等待的是 server 方的报文队列中出现空间而使报文得以投递，可是那并不意味着操作的完成，本次系统调用要到连接请求被接受时才会完成。显然，那并不是在一个有限的、可以预测的短时间内能够完成的，所以只好使系统调用提前结束，先来处理已经到达的信号。

过了这一关，目标插口，即 server 插口一方已经没有问题了，但是 client 插口自己这一方呢？我们继续往下看函数 unix_stream_connect( )的代码（af_unix.c）。

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( )]

```
927		/* Latch our state.
928	
929		   It is tricky place. We need to grab write lock and cannot
930		   drop lock on peer. It is dangerous because deadlock is
931		   possible. Connect to self case and simultaneous
932		   attempt to connect are eliminated by checking socket
```

```
933             state. other is TCP_LISTEN, if sk is TCP_LISTEN we
934             check this before attempt to grab lock.
935
936             Well, and we have to recheck the state after socket locked.
937          */
938          st = sk->state;
939
940          switch (st) {
941          case TCP_CLOSE:
942              /* This is ok... continue with connect */
943              break;
944          case TCP_ESTABLISHED:
945              /* Socket is already connected */
946              err = -EISCONN;
947              goto out_unlock;
948          default:
949              err = -EINVAL;
950              goto out_unlock;
951          }
952
953          unix_state_wlock(sk);
954
955          if (sk->state != st) {
956              unix_state_wunlock(sk);
957              unix_state_runlock(other);
958              sock_put(other);
959              goto restart;
960          }
961
```

代码中 940 行的 switch 语句对 client 方插口的状态进行检查，看看是否处于 TCP_CLOSE 状态。读者也许要问，这个检查为什么不放在 sys_connect( )或者 unix_stream_connect( )一开头的地方呢？那样如果状态不对的话一开始就可以回头了，岂不是可以省去这么多麻烦？答案是，即使在一开始时就作了这种检查，现在也还得再检查，因为情况可能中途有改变。要知道，可能会有多个进程（创建了插口的进程可能会 fork( )出一批子进程）并发地对同一个插口发动操作，从而让另一个进程抢了先。另一方面，一个 client 方的进程也不能一开始就把 sock 结构锁住不放，因为在 unix_stream_connect( )中可能会进入睡眠（否则就有可能导致死锁了，请读者想想为什么？）所以，不管怎样，在这个地方总是需要对 client 插口的状态进行检查的。通过了检查以后，就要改变其状态了，所以调用 unix_state_wlock( )将其锁住（见 952 行）。可是在加锁成功以后还得再检查一次（见 955 行）！为什么呢？须知 unix_state_wlock( )中可能隐藏着一个循环等待，而引起循环等待的原因是另一个进程（在另一个 CPU 上运行）抢先把它锁住了，所以在循环等待的前后这个 sock 结构的状态可能就不一样了。这一段代码（938～960 行，直到结束）看似平常，实际上却极有讲究，对于进程间的同步与互斥是一段很好的教材，建议读者把它读透，并且多问几个为什么，再自己来解答。

接着，就要来设置有关的数据结构了（af_unix.c）：

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( )]

```
962         /* The way is open! Fastly set all the necessary fields... */
963
964         sock_hold(sk);
965         unix_peer(newsk)=sk;
966         newsk->state=TCP_ESTABLISHED;
967         newsk->type=SOCK_STREAM;
968         newsk->peercred.pid = current->pid;
969         newsk->peercred.uid = current->euid;
970         newsk->peercred.gid = current->egid;
971         newsk->sleep = &newsk->protinfo.af_unix.peer_wait;
972
973         /* copy address information from listening to new sock*/
974         if (other->protinfo.af_unix.addr)
975         {
976             atomic_inc(&other->protinfo.af_unix.addr->refcnt);
977             newsk->protinfo.af_unix.addr=other->protinfo.af_unix.addr;
978         }
979         if (other->protinfo.af_unix.dentry) {
980         newsk->protinfo.af_unix.dentry = dget(other->protinfo.af_unix.dentry);
981         newsk->protinfo.af_unix.mnt= mntget(other->protinfo.af_unix.mnt);
982         }
983
984         /* Set credentials */
985         sk->peercred = other->peercred;
986
987         sock_hold(newsk);
988         unix_peer(sk)=newsk;
989         sock->state=SS_CONNECTED;
990         sk->state=TCP_ESTABLISHED;
991
992         unix_state_wunlock(sk);
993
```

这里涉及的数据结构有这么几个：指针 sock 指向 client 插口的 socket 结构，指针 sk 指向 client 插口的 sock 结构，指针 other 指向 server 插口的 socket 结构，而指针 newsk 则指向一个新的 sock 结构。这个新的 sock 结构准备传给 server 方进程，以供在 accept( )中新创建的插口配对使用。这里的 unix_peer( )是个宏操作，其定义在文件 af_unix.c 中：

```
140     #define unix_peer(sk) ((sk)->pair)
```

所以，经过这段程序以后，newsk->pair 指向 sk，而 sk->pair 指向 newsk（这就是 Unix 域中插口间连接的主体）。现在二者已经挂上了钩，建立起连接，所缺少的环节就是让 newsk 跟 server 方进程在 accept( )中新创建的插口挂上钩了。另一方面，client 方插口的 sock 结构的状态也变成了 TCP_ESTABLISHED。这样，如果有另一个进程也要在这个插口上进行 connect( )操作，就会在上面的

944 行受到阻拦而失败返回。

代码中 980 行的 dget( )实际上只是递增 dentry 结构中的共享计数 d_count，我们已经在“文件系统”一章中看到过了。981 行的 mntget( )也类似。

我们尚未将连接请求报文，即 sk_buff 结构挂入 server 方 sock 结构的 receive_queue 队列中（代码中的指针 skb 指向这个结构），下面就要来做这件事了。继续在 af_unix.c 中往下看：

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( )]

```
 994         /* take ten and and send info to listening sock */
 995         skb_queue_tail(&other->receive_queue, skb);
 996         unix_state_runlock(other);
 997         other->data_ready(other, 0);
 998         sock_put(other);
 999         return 0;
1000
1001     out_unlock:
1002         if (other)
1003             unix_state_runlock(other);
1004
1005     out:
1006         if (skb)
1007             kfree_skb(skb);
1008         if (newsk)
1009             unix_release_sock(newsk, 0);
1010         if (other)
1011             sock_put(other);
1012         return err;
1013     }
```

这里 skb_queue_tail( )是个 inline 函数，其代码在 include/linux/skbuff.h 中：

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( ) > skb_queue_tail( )]

```
463    /**
464     *  skb_queue_tail - queue a buffer at the list tail
465     *  @list: list to use
466     *  @newsk: buffer to queue
467     *
468     *  Queue a buffer at the tail of the list. This function takes the
469     *  list lock and can be used safely with other locking &sk_buff functions
470     *  safely.
471     *
472     *  A buffer cannot be placed on two lists at the same time.
473     */
474
475    static inline void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk)
```

```
476    {
477        unsigned long flags;
478
479        spin_lock_irqsave(&list->lock, flags);
480        __skb_queue_tail(list, newsk);
481        spin_unlock_irqrestore(&list->lock, flags);
482    }
```

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( ) > skb_queue_tail( ) > __skb_queue_tail( )]

```
449    static inline void __skb_queue_tail(struct sk_buff_head *list,
                          struct sk_buff *newsk)
450    {
451        struct sk_buff *prev, *next;
452
453        newsk->list = list;
454        list->qlen++;
455        next = (struct sk_buff *)list;
456        prev = next->prev;
457        newsk->next = next;
458        newsk->prev = prev;
459        next->prev = newsk;
460        prev->next = newsk;
461    }
```

最后，在 sock 结构中有个函数指针 data_ready，每当将一个报文（无论控制报文或是数据报文）链入到一个 sock 结构的 receive_queue 队列中以后，都要通过这个函数指针来调用一个预先设置好的函数（997 行），这种情况常称为“call back”。在创建插口时所调用的函数 sock_init_data( )里，函数指针 data_ready 被设置成指向 sock_def_readable( )，所以这里的“call back”就是对这个函数的调用，其代码在 net/core/sock.c 中：

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( ) > sock_def_readable( )]

```
1090   void sock_def_readable(struct sock *sk, int len)
1091   {
1092       read_lock(&sk->callback_lock);
1093       if (sk->sleep && waitqueue_active(sk->sleep))
1094           wake_up_interruptible(sk->sleep);
1095       sk_wake_async(sk,1,POLL_IN);
1096       read_unlock(&sk->callback_lock);
1097   }
```

对于此时要做的事情，读者大概至少已经猜对了一半，那就是要唤醒可能正在睡眠中等待着连接请求的 server 方进程（1094 行）。但是，另一半，也就是 sock_wake_async( )，是干什么用的呢？让我们回顾一下 server 方进程是怎样通过 accept( )来接受连接请求的。大家知道，accept( )是一个 server 插口接受连接请求的惟一途径，server 插口是不能主动要求连接的。同时，accept( )的操作从本质上说是

“同步”的，如果调用 accept( )时尚无连接请求到来，就要睡眠等待。诚然，server 方的进程可以通过 O_NONBLOCK 标志让 accept( )在没有连接请求时立即返回，但这样一来，server 方进程就只好循坏地或者定期地调用 accept( )来测试是否有连接请求到来。再考虑有时候 server 方进程要同时照看好几个 server 插口的情况，这时候 server 方进程就只好将 O_NONBLOCK 标志设成 1 来“轮询”各个 server 插口了。但是，不管是睡眠也好，或是轮询也好，server 进程在此期间就不能做别的事情了。所以，O_NONBLOCK 标志并不改变 accept( )的同步本质。

那么，有没有办法“异步”地等待连接呢？就是说使 server 方进程可以干点别的，到有连接请求到来时就通知它，让它到那时候再来调用 accept( )。答案是肯定的。我们以前讲过，硬件（包括处理器）层次上的异步通信手段是中断，而软件层次上的异步通信手段是“信号”。显然，我们在这里也可以利用信号机制。简而言之，就是让内核在有连接请求到来时就向 server 方进程发一个信号，而 server 方进程则平时可以处理别的事情，只是在接收到有关信号时就来调用 accept( )。事实上，并不是只有插口的连接才会有这样的要求，异步操作已经成为文件系统操作的一个组成部分。在文件操作 ioctl( )中就设置了一条命令 FIOASYNC，让有关进程（必须是文件的主人）可以通过这条命令向一个文件挂上号，让它在某种条件得到满足时就向该进程发送一个信号。为了这个目的，在插口的 socket 结构中设置了一个队列 fasync_list。当 server 方进程希望异步地等待连接时，就通过 ioctl( )的 FIOASYNC 命令（插口也是一个已打开文件）将一个 fasync_struct 结构挂入到这个队列中。另一方面，我们以前也讲过，代表着一个插口的 file 结构中有个指针 f_op，指向一个 file_operations 结构 socket_file_ops。这个结构中的指针 fasync 指向函数 sock_fasync( )，当 server 方进程通过 ioctl( )发出 FIOASYNC 命令时，就会执行这个函数来完成上述将一个 fasync_struct 数据结构挂入到这个队列中的操作。

所以，在“call back”函数 sock_def_readable( )中的另一件事就是：通过 sk_wake_async( )给可能已经在队列 fasync_list 中挂上号的进程发信号，其代码在 include/net/sock.h 中：

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( ) > sock_def_readable( ) > sk_wake_async( )]

```
1219    static inline void sk_wake_async(struct sock *sk, int how, int band)
1220    {
1221        if (sk->socket && sk->socket->fasync_list)
1222            sock_wake_async(sk->socket, how, band);
1223        }
```

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( ) > sock_def_readable( ) > sk_wake_async( ) > sock_wake_async( )]

```
786    /* This function may be called only under socket lock or callback_lock */
787
788    int sock_wake_async(struct socket *sock, int how, int band)
789    {
790        if (!sock || !sock->fasync_list)
791            return -1;
792        switch (how)
793        {
794        case 1:
795
```

```
796             if (test_bit(SOCK_ASYNC_WAITDATA, &sock->flags))
797                 break;
798             goto call_kill;
799         case 2:
800             if (!test_and_clear_bit(SOCK_ASYNC_NOSPACE, &sock->flags))
801                 break;
802             /* fall through */
803         case 0:
804         call_kill:
805             __kill_fasync(sock->fasync_list, SIGIO, band);
806             break;
807         case 3:
808             __kill_fasync(sock->fasync_list, SIGURG, band);
809         }
810         return 0;
811     }
```

这里调用时的参数 how 为 1，band 为 POLL_IN。我们把这段代码留给读者自己阅读。其中函数 __kill_fasync( )的代码在 fs/fcntl.c 中，它扫描整个 fasync_list 队列，向每个挂上号的进程发出信号。

[sys_socketcall( ) > sys_connect( ) > unix_stream_connect( ) > sock_def_readable( ) > sk_wake_async( ) > sock_wake_async( ) > __kill_fasync( )]

```
482     void __kill_fasync(struct fasync_struct *fa, int sig, int band)
483     {
484         while (fa) {
485             struct fown_struct * fown;
486             if (fa->magic != FASYNC_MAGIC) {
487                 printk(KERN_ERR "kill_fasync: bad magic number in "
488                        "fasync_struct!\n");
489                 return;
490             }
491             fown = &fa->fa_file->f_owner;
492             /* Don't send SIGURG to processes which have not set a
493                queued signum: SIGURG has its own default signalling
494                mechanism. */
495             if (fown->pid && !(sig == SIGURG && fown->signum == 0))
496                 send_sigio(fown, fa->fa_fd, band);
497             fa = fa->fa_next;
498         }
499     }
```

不言而喻，如果要异步地接受连接请求，则 server 方的进程必须事先设置好相应的信号处理程序。

至此，sys_connect( )已经完成了它的任务（除 sockfd_put( )以外）。如果 server 方进程已经在 accept( )中等待，则唤醒以后就会来补上缺失的一环，就是使新创建的 socket 结构与通过 sk_buff 结构传过来的 sock 结构挂上钩（见 unix_accept( )代码中的 1068 行，注意那里的指针 newsock 指向由 accept( )新创建

的 socket 结构，而指针 tsk 指向由 sys_connect( )新创建的 sock 结构)。这样，就完成了两个 unix 域插口之间的连接。如果 server 方进程尚未调用 accept( )，则当它调用 accept( )时连接请求已经在 server 插口的 receive_queue 队列中等待，所以无需睡眠等待就可以完成这种连接。

再来看“无连接”模式的 connect( )操作。对于 Unix 域“无连接”模式的插口，所用的 proto_ops 数据结构为 unix_dgram_ops，而相应的函数指针 connect 则指向 unix_dgram_connect( )。

以前讲过，既然是“无连接”模式的插口，本来就没有“建立连接”这一说。之所以也有个 connect( )操作，只是要利用它来将本来每次发送报文时都要重复的一些操作集中在一起，以避免浪费。有些什么“每次发送报文时都要重复的操作”呢？

(1) 每次都要从用户空间把对方的插口地址拷贝到系统空间中。

(2) 在网络环境下，通常需要根据对方的地址从路径表中查得应该使用的网络接口，以及可能需要的在网络层和 / 或链路层上使用的地址。并且常常还要进行从符号地址到网络地址数值的转换。

(3) 在 Unix 域中，则通常需要根据对方的插口地址从文件系统中打开相应的文件（节点），然后用索引节点的号码在杂凑表的某个队列中找到对方的 sock 数据结构。

就每一次报文发送来说，这些开销不能算大。但是，如果发送成千上万个报文到同一个目标插口，这总共的开销就不能小看了。下面，我们就来看看，sys_connect( )对于 Unix 域“无连接”模式的插口到底做了些什么。函数 unxi_dgram_connect( )的代码在 net/unix/af_unix.c 中：

[sys_socketcall( ) > sys_connect( ) > unix_dgram_connect( )]

```
770    static int unix_dgram_connect(struct socket *sock, struct sockaddr *addr,
771                    int alen, int flags)
772    {
773        struct sock *sk = sock->sk;
774        struct sockaddr_un *sunaddr=(struct sockaddr un*)addr;
775        struct sock *other;
776        unsigned hash;
777        int err;
778
779        if (addr->sa_family != AF_UNSPEC) {
780            err = unix_mkname(sunaddr, alen, &hash);
781            if (err < 0)
782                goto out;
783            alen = err;
784
785            if (sock->passcred && !sk->protinfo.af_unix.addr &&
786                (err = unix autobind(sock)) != 0)
787                goto out;
788
789            other=unix_find_other(sunaddr, alen, sock->type, hash, &err);
790            if (!other)
791                goto out;
792
793            unix_state_wlock(sk);
```

```
794
795             err = -EPERM;
796             if (!unix_may_send(sk, other))
797                 goto out_unlock;
798         } else {
799             /*
800              *  1003.1g breaking connected state with AF_UNSPEC
801              */
802             other = NULL;
803             unix_state_wlock(sk);
804         }
805
806         /*
807          * If it was connected, reconnect.
808          */
809         if (unix_peer(sk)) {
810             struct sock *old_peer = unix_peer(sk);
811             unix_peer(sk)=other;
812             unix_state_wunlock(sk);
813
814             if (other != old_peer)
815                 unix_dgram_disconnected(sk, old_peer);
816             sock_put(old_peer);
817         } else {
818             unix_peer(sk)=other;
819             unix_state_wunlock(sk);
820         }
821         return 0;
822
823     out_unlock:
824         unix_state_wunlock(sk);
825         sock_put(other);
826     out:
827         return err;
828     }
```

读者已经学习过“有连接”模式插口的 accept( )和 connect( )操作的代码，对于这里所调用或引用的大部分函数和宏定义都已经熟悉了。事实上，读者在前几节中尚未见到过的函数只有一个，那就是 unix_may_send( )，其代码也在 af_unix.c 中：

[sys_socketcall( ) > sys_connect( ) > unix_dgram_connect( ) > unix_may_send( )]

```
140     #define unix_peer(sk) ((sk)->pair)
141
142     extern __inline    int unix_our_peer(unix_socket *sk, unix socket *osk)
143     {
144         return unix peer(osk) == sk;
```

```
145     }
146
147     extern __inline__ int unix_may_send(unix_socket *sk, unix_socket *osk)
148     {
149         return (unix_peer(osk) == NULL || unix_our_peer(sk, osk));
150     }
```

也就是说，unix_may_send( )检查对方插口的 sock 结构中的指针 pair，看看它是否已经指向某个 sock 结构，如果是，就进而检查它所指向的是否正是我方插口的 sock 结构。如果对方 sock 结构中的指针 pair 已经指向了另一个插口的 sock 结构，那就只好就此打住了。这种情况下，unix_dgram_connect( ) 返回出错代码－EPERM，表示不允许。也就是说，对方插口必须尚未连接到别的插口，改变对方插口与其他插口的连接是不允许的。可是，如果我方插口原来已经有了“连接”，则允许以新的连接代替原有的连接（见 809～816 行），而不需要特别将原有的连接拆除。这里所谓“连接”只不过是让我方 sock 结构中的指针 pair 指向对方的 sock 结构。注意，这个“连接”只是单向的，对方插口的数据结构丝毫不受影响，对方进程有完全的自由使其插口指向（或曰“连接到”）其他插口。此外，“无连接” 插口也不像“有连接”插口那样有个“状态机”，所以“无连接”通信有时也称作“无状态”通信；对方插口也不会像“有连接”模式的 server 插口那样生下一个“蛋”来。正因为这样，unix_dgram_connect( ) 的代码比 unix_stream_connect( )的代码要简单多了。

7.7　报文的接收与发送

插口上的报文接收与发送有两个程序设计界面。第一个界面是为插口专设的，从用户程序的角度来看就是三对 libc 库函数，即 recv( )/send( )、recvfrom( )/sendto( )以及 recvmsg( )/sendmsg( )，但是最终都归结于一个统一的系统调用，在内核中的入口为 sys_socketcall( )。不过，除特殊的应用外（后面会讲到），这些库函数并不是非得成对地使用。也就是说，原则上双方都可以自由地选择使用二者之一，但是 send( )只能在已经通过 connect( )建立了连接，或确定了目标插口以后才可使用。另一方面，从语义的角度来说，只要已经使用了 connect( )就应该用 send( )而不是 sendto( )，因为既然目标已经确定了，就不应该再在发送时规定目标了。不过，尽管如此，用 sendto( )也还是可以的，只不过要把调用参数中的对方地址（指针）设成 NULL，把地址长度设成 0。读者在前面已经看到过，在内核中 sys_recv( ) 实际上就是通过 sys_recvfrom( )实现的，而 sys_send( )则是通过 sys_sendto( )实现的，无非就是把地址指针设成 NULL，把地址长度设成 0 而已。此外，下面读者还会看到，实际上三个用于接收的函数最后全都通过 sock_recvmsg( )来接收报文，而三个用于发送的函数则全都是通过 sock_sendmsg( )来发送报文的。

第二个界面是通过常规的文件操作 read( )和 write( )这两对系统调用来进行的（还有 readv( )和 writev( )，与 recvmsg( )和 sendmsg( )相似）。读者不妨回过去看看前面的联系图（图 7.1）。从当前进程的 task_struct 结构开始，通过插口的打开文件号找到相应的 file 结构；再顺着 file 结构中的指针 f_op，就可以找到这个“文件”的 file_operations 结构 socket_file_ops。这就是文件操作的跳转表，是在 socket.c 中定义的：

```
114     static struct file_operations socket_file_ops = {
115         llseek:     sock_lseek,
```

```
116         read:        sock_read,
117         write:       sock_write,
118         poll:        sock_poll,
119         ioctl:       sock_ioctl,
120         mmap:        sock_mmap,
121         open:        sock_no_open,    /* special open code to disallow open via /proc */
122         release:     sock_close,
123         fasync:      sock_fasync,
124         readv:       sock_readv,
125         writev:      sock_writev
126     };
```

由此可见，相应的函数为 sock_read( )和 sock_write( )，这两个函数的代码都在 net/socket.c 中。这里要提醒一下，插口一经创建就已经打开了（返回打开文件号），并没有一般文件那样的 open( )操作，所以指针 open 指向一个函数 sock_no_open( )。

函数 sock_write( )与 sys_sendto( )很相似，我们不妨比较一下 sock_write( )和 sys_sendto( )的代码。先看 sock_write( )：

[sys_write( ) > sock_write( )]

```
571     /*
572      *  Write data to a socket. We verify that the user area ubuf..ubuf+size-1
573      *  is readable by the user process.
574      */
575
576     static ssize_t sock_write(struct file *file, const char *ubuf,
577                   size_t size, loff_t *ppos)
578     {
579         struct socket *sock;
580         struct msghdr msg;
581         struct iovec iov;
582
583         if (ppos != &file->f_pos)
584             return -ESPIPE;
585         if(size==0)       /* Match SYS5 behaviour */
586             return 0;
587
588         sock = socki_lookup(file->f_dentry->d_inode);
589
590         msg.msg_name=NULL;
591         msg.msg_namelen=0;
592         msg.msg_iov=&iov;
593         msg.msg_iovlen=1;
594         msg.msg_control=NULL;
595         msg.msg_controllen=0;
596         msg.msg_flags=!(file->f_flags & O_NONBLOCK) ? 0 : MSG_DONTWAIT;
597         if (sock->type == SOCK_SEQPACKET)
```

```
598             msg.msg_flags |= MSG_EOR;
599         iov.iov_base=(void *)ubuf;
600         iov.iov_len=size;
601
602         return sock_sendmsg(sock, &msg, size);
603     }
```

再看 sys_sendto( ):

[sys_socketcall( ) > sys_sendto( )]

```
1159    /*
1160     * Send a datagram to a given address. We move the address into kernel
1161     * space and check the user space data area is readable before invoking
1162     * the protocol.
1163     */
1164
1165    asmlinkage long sys_sendto(int fd, void * buff, size_t len, unsigned flags,
1166                    struct sockaddr *addr, int addr_len)
1167    {
1168        struct socket *sock;
1169        char address[MAX_SOCK_ADDR];
1170        int err;
1171        struct msghdr msg;
1172        struct iovec iov;
1173
1174        sock = sockfd_lookup(fd, &err);
1175        if (!sock)
1176            goto out;
1177        iov.iov_base=buff;
1178        iov.iov_len=len;
1179        msg.msg_name=NULL;
1180        msg.msg_iov=&iov;
1181        msg.msg_iovlen=1;
1182        msg.msg_control=NULL;
1183        msg.msg_controllen=0;
1184        msg.msg_namelen=addr_len;
1185        if(addr)
1186        {
1187            err = move_addr_to_kernel(addr, addr_len, address);
1188            if (err < 0)
1189                goto out_put;
1190            msg.msg_name=address;
1191        }
1192        if (sock->file->f_flags & O_NONBLOCK)
1193            flags |= MSG_DONTWAIT;
1194        msg.msg_flags = flags;
```

```
1195        err = sock sendmsg(sock, &msg, len);
1196
1197    out_put:
1198        sockfd_put(sock);
1199    out:
1200        return err;
1201    }
```

可见二者几乎是一样的，最后都是调用 sock_sendmsg( )来完成任务。同样的相似性也存在于 sock_read( )和 sys_recvmsg( )之间，并且二者都通过 sock_recvmsg( )来完成任务。也就是说，两个界面上的这些函数最后都是殊途同归，都归结到 sock_recvmsg( )和 sock_sendmsg( )两个函数。所以，两个界面其实没有多大区别。不过，从语义的角度来说，一般对“有连接”插口倾向于使用 read( )/write( )，而对“无连接”插口则通常都使用 recvfrom( )/sendto( )等函数。这是因为在“有连接”模式的通信中将传递的数据看成连续的“字节流”，而不保留“报文”的边界（所以其类型称为 SOCK_STREAM），与文件操作的语义比较贴近。反之，“无连接”模式的通信则是“面向报文”的，所以保留报文的边界。

我们以接收方为例进一步说明这两种模式间的区别。假定在一个“无连接”插口的 receive_queue 队列中已经有两个报文在等待读取，每个报文都含有 200 字节的数据。然后，接收方进程通过 recv( ) 来接收，缓冲区的大小为 150 字节。由于缓冲区小于队列中第一个报文的大小，所以只能接收 150 个字节，于是 recv( )返回 150，即实际接收到的字节数。此时虽然第一个报文中尚有 50 字节的剩余，但是内核却会将其从队列中脱链并释放，剩余的 50 字节就丢弃了。这是因为“无连接”模式的通信是以报文为单位，而不是以实际报文中的字节为单位的。反之，如果缓冲区的大小是 300 字节，那么虽然缓冲区大于第一个报文的实际大小，却不会再到第二个报文中去读取一部分，以填满缓冲区，所以recv( )返回 200。可是，如果插口是“有连接”模式的，那就不同了。在第一种情况中，剩余的 50 字节会保留下来供下一次继续读取，而在第二种情况下则会在读取了第一个报文中的 200 字节之后再到第二个报文中读取 100 字节，而将第二个报文中剩余的 100 字节留待下一次继续读取，所以此时 recv( )返回 300。当然，“有连接”和“无连接”两种通信模式的区别远不止于此。但是，对 Unix 域的插口来说，可靠性的问题实际上并不存在（因为不涉及网络介质），保序性的问题也不存在（因为不涉及不同的路径），剩下的就是连接的建立以及语义上的这种区别了。从用户程序设计的角度来看，后者实际上更为重要。举例来说，如果用户程序未经建立连接就试图通过一个“有连接”插口发送信息，那么 send( ) 马上就会出错返回，从而一开始就可以发现这个问题。然而，如果是在“无连接”插口上用 read( )来接收，并且在程序设计中误以为接收的内容是连续的字节流，则系统调用本身并不会出错返回，可是却会出现一些似乎是莫明其妙的毛病而又不容易找到原因。反过来，将互相独立的报文作为连续的字节流来接收也会造成问题。

在深入到 sock_recvmsg( )和 sock_sendmsg( )的代码中去之前，还要先回到前面 sys_sendto( )的代码中去看一下这两个函数的外围。由于这两个函数要适应上述所有函数的需要，所以是按其中最复杂的 sys_sendmsg( )和 sys_recvmsg( )的要求而设计的。因此，在 sys_sendto( )中要先使用传递下来的参数，组装起一个“报头”，即 msghdr 结构，用作调用 sock_sendmsg( )的主要参数。有关定义见 include/linux/socket.h：

```
27    /*
28     *  As we do 4.4BSD message passing we use a 4.4BSD message passing
29     *  system, not 4.3. Thus msg_accrights(len) are now missing. They
```

```
30      *  belong in an obscure libc emulation or the bin.
31      */
32
33     struct msghdr {
34         void    *       msg_name;       /* Socket name          */
35         int             msg_namelen;        /* Length of name       */
36         struct iovec *  msg_iov;            /* Data blocks          */
37         __kernel_size_t msg_iovlen;     /* Number of blocks     */
38         void            *msg_control;   /* Per protocol magic (eg BSD file
                                                descriptor passing) */
39         __kernel_size_t msg_controllen; /* Length of cmsg list */
40         unsigned        msg_flags;
41     };
42
43     /*
44      *  POSIX 1003.1g - ancillary data object information
45      *  Ancillary data consits of a sequence of pairs of
46      *  (cmsghdr, cmsg_data[ ])
47      */
48
49     struct cmsghdr {
50         __kernel_size_t cmsg_len;   /* data byte count, including hdr */
51             int     cmsg_level; /* originating protocol */
52             int     cmsg_type;  /* protocol-specific type */
53     };
```

这里msghdr结构中的msg_name、msg_namelen以及msg_flags分别对应于sys_sendto( )的参数addr、addr_len以及flags。指针msg_control可以指向一个附加的数据结构，用来提供一些附加的控制信息，其类型取决于具体的规程，一般为cmsghdr数据结构加cmsg_data[ ]数组。此外，更重要的是，结构中有个指针msg_iov指向一个iovec结构数组，其定义在include/linux/uio.h中，而msg_iovlen则为该数组的大小：

```
19     struct iovec
20     {
21       void      *iov_base;  /* BSD uses caddr_t (1003.1g requires void *) */
22       __kernel_size_t iov_len; /* Must be size_t (1003.1g) */
23     };
```

数组中的每一个元素即iovec结构，都是一个所谓“io向量”，由指向数据缓冲区的指针iov_base和表示缓冲区中数据长度的iov_len构成。这样，由报头msghdr结构所代表的报文可以由多个数据缓冲区构成，还可以包含由msg_control所指向的附加信息（通常是控制信息）。上述各个数据结构之间的联系如图7.3所示。

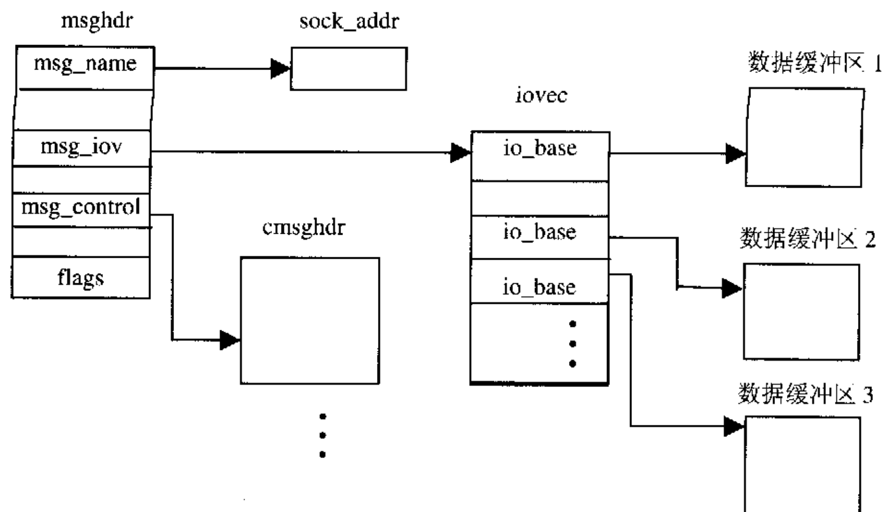


图 7.3　报文数据结构联系图

所以，msghdr 结构代表了报文的最为一般的形式。当报文中只有一个“io 向量”，即只有一个数据缓冲区，并且不含有控制信息时，就相当于普通的“缓冲区地址加数据长度”形式。那么，为什么在报文的一般形式中需要能容纳多个数据缓冲区呢？让我们考虑一下在网络环境下的报文接收，就以 Ethernet 为例吧。在 Ethernet 中，一个报文（packet，或 frame）的最大长度约为 1500 字节，而最小长度只有数十字节。当开始从网络接收报文时，其长度通常是未知的，要到接收完毕时才能知道它的长度。另一方面，为了提高效率，用于从网络上接收报文的缓冲区都是预先分配好在一个“缓冲池”中备用的，缓冲池中所有的缓冲区都具有相同的长度。可是，每个缓冲区的大小以什么为准呢？如果都按可能的最大长度分配，那就必然造成很大的浪费，因为实际上在 Ethernet 中传递的报文多数是 100 字节以下的。如果减少缓冲区长度，那就势必有时候得用好几个缓冲区才能容纳一个报文。由此可见，比较合理的安排就是类似于 msghdr 数据结构那样的设计，选择适中的缓冲区长度，以多个缓冲区来容纳一个报文，允许在最后一个缓冲区中浪费少量的空间。不过要指出，这里讲的使用多个缓冲区是指在接收进程（或发送进程）与插口之间，而不是指插口与插口之间的报文缓冲区。

看了 sys_sendto( )，如何将其参数组装成一个以 msghdr 结构为代表的报文的，则 sys_recvfrom( ) 如何将其还原就是不言而喻的了。而我们在前面已经看到过 sys_send( )和 sys_recv( )分别是通过 sys_sendto( )和 sys_recvfrom( )实现的，只是将参数中的指针 addr 设成 NULL，整数 addrlen 设成 0 而已。至于 read( )和 write( )，我们已经看到其与 sock_write( )和 sys_sendto( )的相似性。

现在可以来看 sock_recvmsg( )和 sys_recvfrom( )的代码。先看前者（net/socket.c）：

[sys_socketcall( ) > sys_recvmsg( )]

```
514	int sock_recvmsg(struct socket*sock, struct msghdr *msg, int size, int flags)
515	{
516		struct scm cookie scm;
517
518		memset(&scm, 0, sizeof(scm));
519
```

```
520         size = sock->ops->recvmsg(sock, msg, size, flags, &scm);
521         if (size >= 0)
522             scm_recv(sock, msg, &scm, flags);
523
524         return size;
525     }
```

对于 Unix 域，根据插口类型的不同，具体的函数可以是 unix_dgram_recvmsg ( )或者 unix_stream_recvmsg( )。

再看 sys_recvfrom( ):

[sys_socketcall( ) > sys_recvfrom( )]

```
1212    /*
1213     * Receive a frame from the socket and optionally record the address of the
1214     * sender. We verify the buffers are writable and if needed move the
1215     * sender address from kernel to user space.
1216     */
1217
1218     asmlinkage long sys_recvfrom(int fd, void * ubuf, size_t size, unsigned flags,
1219                     struct sockaddr *addr, int *addr_len)
1220    {
1221        struct socket *sock;
1222        struct iovec iov;
1223        struct msghdr msg;
1224        char address[MAX_SOCK ADDR];
1225        int err, err2;
1226
1227        sock = sockfd_lookup(fd, &err);
1228        if (!sock)
1229            goto out;
1230
1231        msg.msg_control=NULL;
1232        msg.msg_controllen=0;
1233        msg.msg_iovlen=1;
1234        msg.msg_iov=&iov;
1235        iov.iov_len=size;
1236        iov.iov_base=ubuf;
1237        msg.msg name=address;
1238        msg.msg namelen=MAX_SOCK_ADDR;
1239        if (sock->file->f_flags & O_NONBLOCK)
1240            flags |= MSG DONTWAIT;
1241        err=sock_recvmsg(sock, &msg, size, flags);
1242
1243        if(err >= 0 && addr != NULL && msg.msg_namelen)
1244        {
1245            err2=move_addr_to_user(address, msg.msg_namelen, addr, addr_len);
1246            if(err2<0)
```

```
1247                    err=err2;
1248            }
1249            sockfd_put(sock);
1250        out:
1251            return err;
1252        }
```

可见，实际上 sys_recvfrom( )的主体就是 sock_recvmsg( )，其代码在 net/socket.c 中：

[sys_socketcall( ) > sys_recvfrom( ) > sock_recvmsg( )]

```
514   int sock_recvmsg(struct socket *sock, struct msghdr *msg, int size, int flags)
515     {
516         struct scm_cookie scm;
517
518         memset(&scm, 0, sizeof(scm));
519
520         size = sock->ops->recvmsg(sock, msg, size, flags, &scm);
521         if (size >= 0)
522             scm_recv(sock, msg, &scm, flags);
523
524         return size;
525     }
```

这个函数的代码与上面的 sys_recvmsg( )一模一样。简单地说，这个函数就做两件事。第一件是接收报文中的数据以及附加信息，第二件就是对附加信息的处理。

接收附加信息要用到一个数据结构，那就是 scm_cookie 结构，是在 include/net/scm.h 中定义的：

```
4     /* Well, we should have at least one descriptor open
5      * to accept passed FDs 8)
6      */
7     #define SCM_MAX_FD  (OPEN_MAX-1)
8
9     struct scm_fp_list
10    {
11        int      count;
12        struct file *fp[SCM_MAX_FD];
13    };
14
15    struct scm_cookie
16    {
17        struct ucred        creds;      /* Skb credentials  */
18        struct scm_fp_list  *fp;        /* Passed files     */
19        unsigned long       seq;        /* Connection seqno */
20    };
```

从数据结构的定义可以看出，这里主要有两种信息。一种是对方进程的“身份”（credentials）。数

据结构类型 ucred 是在 include/linux/socket.h 中定义的：

```
125    struct ucred {
126        __u32   pid;
127        __u32   uid;
128        __u32   gid;
129    };
```

另一种就是有关打开文件的信息。BSD（以及 Linux）的插口机制允许一个进程把它的若干个已打开文件的访问权传递给，或者说"授予"另一个进程。一个进程最多可以同时打开的文件个数取决于一个常数 OPEN_MAX，定义于 include/linux/limits.h 中：

```
9  #define OPEN_MAX    256        /* # open files a process may have */
```

可是，有一个位置是必须保留的，那就是代表着插口本身的那个已打开文件。所以在 scm_fd_list 结构中的 file 结构指针数组 fp[ ]的大小为（OPEN_MAX－1）。

首先是数据部分的接收。对于 Unix 域的插口，sock->ops 视插口类型的不同而指向 unix_dgram_ops 或 unix_stream_ops，二者的 recvmsg 指针分别指向 unix_dgram_recvmsg( )和 unix_stream_recvmsg( )。

我们先来看比较简单的 unix_dgram_recvmsg( )（af_unix.c）：

[sys_socketcall( ) > sys_recvmsg( ) > unix_dgram_recvmsg( )]

```
1388    static int unix_dgram_recvmsg(struct socket *sock, struct msghdr *msg, int size,
1399                    int flags, struct scm_cookie *scm)
1400    {
1401        struct sock *sk = sock->sk;
1402        int noblock = flags & MSG_DONTWAIT;
1403        struct sk_buff *skb;
1404        int err;
1405
1406        err = -EOPNOTSUPP;
1407        if (flags&MSG_OOB)
1408            goto out;
1409
1410        msg->msg_namelen = 0;
1411
1412        skb = skb_recv_datagram(sk, flags, noblock, &err);
1413        if (!skb)
1414            goto out;
1415
1416        wake_up_interruptible(&sk->protinfo.af_unix.peer_wait);
1417
1418        if (msg->msg_name)
1419            unix_copy_addr(msg, skb->sk);
1420
1421        if (size > skb->len)
```

```
1422            size = skb->len;
1423        else if (size < skb->len)
1424            msg->msg_flags |= MSG_TRUNC;
1425
1426        err = skb_copy_datagram_iovec(skb, 0, msg->msg_iov, size);
1427        if (err)
1428            goto out_free;
1429
1430        scm->creds = *UNIXCREDS(skb);
1431
1432        if (!(flags & MSG_PEEK))
1433        {
1434            if (UNIXCB(skb).fp)
1435                unix_detach_fds(scm, skb);
1436        }
1437        else
1438        {
1439            /* It is questionable: on PEEK we could:
1440               - do not return fds - good, but too simple 8)
1441               - return fds, and do not return them on read (old strategy,
1442                 apparently wrong)
1443               - clone fds (I choosed it for now, it is the most universal
1444                 solution)
1445
1446                   POSIX 1003.1g does not actually define this clearly
1447                   at all. POSIX 1003.1g doesn't define a lot of things
1448                   clearly however!
1449
1450            */
1451            if (UNIXCB(skb).fp)
1452                scm->fp = scm_fp_dup(UNIXCB(skb).fp);
1453        }
1454        err = size;
1455
1456    out_free:
1457        skb_free_datagram(sk, skb);
1458    out:
1459        return err;
1460    }
```

参数 flags 中有个标志位 MSG_OOB，表示此次接收的目的是所谓“out-of-band”报文。在“有连接”模式的通信中，普通的数据报文在发送时都编上序号，接收时就按次序接收。但是，有些报文是用于控制目的（例如载送着“^c”的报文）而需要优先传递的，此种“编外”报文就称为 OOB 报文。可是，在“无连接”模式中每个报文都是独立的，也无所谓次序，根本不存在 OOB 报文这么个概念，所以对“无连接”插口调用 sock_recvmsg( )时 MSG_OOB 标志位不应该为 1（见 1407 行）。

函数 skb_recv_datagram( )从插口的接收队列中摘取，或者等待着从该队列中摘取一个载运着报文

的 sk_buff 结构，读者在前面讲述 sys_accept( )的一节中已看到过其代码，此处就不重复了。在我们这个情景中，假定队列中已经有报文存在，所以当前进程不需要睡眠等待。从接收队列摘取一个报文以后，队列中就多出了一个报文的位置，所以要唤醒一个可能正在睡眠、等待着要投递报文的进程。如果 msghdr 结构中的指针 msg_name 不是 NULL，也就是说接收进程为对方（发送方）的插口地址准备好了一个缓冲区，就要将对方的插口地址拷贝到一个临时的缓冲区中（见这里的 1419 行和 sys_recvfrom( )代码中的第 1245 行），为下一步将其拷贝到用户空间的缓冲区作好准备。函数 unix_copy_addr( )的代码也在 af_unix.c 中：

[sys_socketcall( ) > sys_recvmsg( ) > unix_dgram_recvmsg( ) > unix_copy_addr( )]

```
1387    static void unix_copy_addr(struct msghdr *msg, struct sock *sk)
1388    {
1389        msg->msg_namelen = sizeof(short);
1390        if (sk->protinfo.af_unix.addr) {
1391            msg->msg_namelen=sk->protinfo.af_unix.addr->len;
1392            memcpy(msg->msg_name,
1393                    sk->protinfo.af_unix.addr->name,
1394                    sk->protinfo.af_unix.addr->len);
1395        }
1396    }
```

下面就是对接收长度的处理了（1421～1424 行）。如果缓冲区的大小，也就是要求接收的数据大小超过报文的实际长度，那就按报文的实际长度接收。反之，如果缓冲区小于报文的实际长度，那就按缓冲区的大小接收，而报文中剩余的数据就丢掉了，所以将 msg_flags 中的 MSG_TRUNC 标志设成 1，表示报文被截尾了。长度一经确定，就可以把数据从 sk_buff 结构中拷贝到由 msghdr 结构中的 iovec[ ] 向量表所指示的缓冲区中，函数 skb_copy_datagram_iovec( )的代码在 net/core/datagram.c 中：

[sys_socketcall( ) > sys_recvmsg( ) > unix_dgram_recvmsg( ) > skb_copy_datagram_iovec ( )]

```
200    /*
201     *  Copy a datagram to an iovec.
202     *  Note: the iovec is modified during the copy.
203     */
204
205     int skb_copy_datagram_iovec(struct sk_buff *skb, int offset, struct iovec *to,
206                     int size)
207    {
208        return memcpy_toiovec(to, skb->h.raw + offset, size);
209    }
```

函数 memcpy_toiovec( )的代码则在 net/core/iovec.c 中：

[sys_socketcall( ) > sys_recvmsg( ) > unix_dgram_recvmsg( ) > skb_copy_datagram_iovec ( )
> memcpy_toiovec( )]

```
76   /*
77    * Copy kernel to iovec. Returns -EFAULT on error.
78    *
79    * Note: this modifies the original iovec.
80    */
81
82   int memcpy_toiovec(struct iovec *iov, unsigned char *kdata, int len)
83   {
84       int err = -EFAULT;
85
86       while(len>0)
87       {
88           if(iov->iov_len)
89           {
90               int copy = min(iov->iov_len, len);
91               if (copy_to_user(iov->iov_base, kdata, copy))
92                   goto out;
93               kdata+=copy;
94               len-=copy;
95               iov->iov_len-=copy;
96               iov->iov_base+=copy;
97           }
98           iov++;
99       }
100      err = 0;
101  out:
102      return err;
103  }
```

对于 Unix 域的插口，一般都只使用一个缓冲区，但是当然也可通过 recvmsg( )来把报文接收到若干个较小的缓冲区中。

除报文中的数据外，还要把 sk_buff 结构中载送的有关对方身份的信息也拷贝到前面准备下的 scm_cookie 结构中去。注意，函数 unix_dgram_recvmsg( )中第 1430 行的赋值语句所做的是整个 ucred 结构的复制，对 UNIXCREDS 的有关定义在 af_unix.h 中：

```
28   struct unix_skb_parms
29   {
30       struct ucred        creds;      /* Skb credentials  */
31       struct scm_fp_list  *fp;        /* Passed files     */
32   };
33
34   #define UNIXCB(skb)      (*(struct unix_skb_parms*)&((skb)->cb))
35   #define UNIXCREDS(skb)   (&UNIXCB((skb)).creds)
```

在 sk_buff 结构中有个 48 字节的字符数组 cb[ ]，可以根据不同网域或不同应用的需要来载送一些附加信息。在这里，用它来传递一个 unix_skb_parms 结构，其内容包括发送方的身份信息以及一个指

向要授权接收方使用的已打开文件表的指针。

至此，报文中的数据都已经拷贝到了用户空间的缓冲区中，附加的信息也拷贝到了临时的 scm_cookie 结构中，从“接收”的角度来说，主要的任务已经完成了。但是，如果 sk_buff 结构中载送着对已打开文件的访问授权的话，那就还有点事要做。当一个进程把对它的若干已打开文件的访问权发送给另一个进程时，要把对这些文件访问权的描绘也接收下来。另一方面，如果这些已打开文件代表着 Unix 域插口，则发送者要为每个这样的已打开文件记下一笔账，说明对它的使用权正在传送的途中。而对方在接收到这些授权后则要负责“销账”。以前讲过，用户进程可以把调用参数 flags 中的 MSG_PEEK 标志置成 1，表示只是“偷看”一下接收队列中的第一个报文，而并不真的接收这个报文。所以 skb_recv_datagram( )在这种情况下并不将报文从队列中摘除，而只是将该报文的共享计数 users 加 1 后便返回指向该 sk_buff 结构的指针。所以，在这两种情况下，对报文中载送的访问授权所作的处理也有所不同。如果接收方在调用 recv( )等函数时的 MSG_PEEK 标志位为 0，也就是说正式“接收”了这些授权时，就要调用 unix_detach_fds( )来销账。这个函数的代码在 af_unix.c 中：

[sys_socketcall( ) > sys_recvmsg( ) > unix_dgram_recvmsg( ) > unix_detach_fds ( )]

```
1111    static void unix_detach_fds(struct scm_cookie *scm, struct sk_buff *skb)
1112    {
1113        int i;
1114
1115        scm->fp = UNIXCB(skb).fp;
1116        skb->destructor = sock_wfree;
1117        UNIXCB(skb).fp = NULL;
1118
1119        for (i=scm->fp->count-1; i>=0; i--)
1120            unix_notinflight(scm->fp->fp[i]);
1121    }
```

这里一方面将 sk_buff 结构中的 scm_fp_list 指针（也在前述的 cb 所载送的 unix_skb_parms 结构中）也拷贝到 scm_cookie 结构中，并为缓冲区的释放准备下一个函数 sock_wfree( )。另一方面就通过 unix_notinflight( )来销账，表示该项授权已不再在“飞行中”。其代码在 net/unix/garbage.c 中：

[sys_socketcall( ) > sys_recvmsg( ) > unix_dgram_recvmsg( ) > unix_detach_fds ( ) > unix_notinflight( )]

```
130     void unix_notinflight(struct file *fp)
131     {
132         unix_socket *s=unix_get_socket(fp);
133         if(s) {
134             atomic_dec(&s->protinfo.af_unix.inflight);
135             atomic_dec(&unix_tot_inflight);
136         }
137     }
```

这里的 unix_get_socket( )从给定的 file 结构出发找到其 inode 结构，如果该 inode 结构代表着一个 Unix 域插口，就返回指向其 sock 结构（inode 结构的一部分）的指针，代码亦在 garbage.c 中：

[sys_socketcall( ) > sys_recvmsg( ) > unix_dgram_recvmsg( ) > unix_detach_fds ( ) > unix_notinflight( ) > unix_get_socket( )]

```
95     extern inline unix_socket *unix_get_socket(struct file *filp)
96     {
97         unix_socket * u_sock = NULL;
98         struct inode *inode = filp->f_dentry->d_inode;
99
100        /*
101         *  Socket ?
102         */
103        if (inode && inode->i_sock) {
104            struct socket * sock = &inode->u.socket_i;
105            struct sock * s = sock->sk;
106
107            /*
108             *  PF_UNIX ?
109             */
110            if (s && sock->ops && sock->ops->family == PF_UNIX)
111                u_sock = s;
112        }
113        return u_sock;
114    }
```

如果接收方只是看一下，而并不真正接收呢？代码的作者在 unix_dgram_recvmsg( )的代码中加了注解，说 POSIX 1003.1g 对此没有作出规定，所以这里采取了为 scm_cookie 结构复制一份 scm_fp_list 结构（而不是其指针）的做法。当然，这时候不能把账销掉，因为对这些已打开文件的访问权仍在递交的途中。

最后，将 sk_buff 结构所占的缓冲区释放掉（1457 行）。

回到 sock_recvmsg( )的代码中（521 行），下一件事就是对接收到的附加信息（如果有的话）的处理了。Inline 函数 scm_recv( )的代码在 include/net/scm.h 中：

[sys_socketcall( ) > sys_recvfrom( ) > sock_recvmsg( ) > scm_recv( )]

```
45     static __inline__ void scm_recv(struct socket *sock, struct msghdr *msg,
46                     struct scm_cookie *scm, int flags)
47     {
48         if (!msg->msg_control)
49         {
50             if (sock->passcred || scm->fp)
51                 msg->msg_flags |= MSG_CTRUNC;
52             scm_destroy(scm);
53             return;
54         }
55
56         if (sock->passcred)
```

```
57          put_cmsg(msg, SOL_SOCKET, SCM_CREDENTIALS, sizeof(scm->creds), &scm->creds);
58
59          if (!scm->fp)
60              return;
61
62          scm_detach_fds(msg, scm);
63      }
```

如果接收进程在其 msghdr 结构中没有为控制信息（附加信息）准备好缓冲区，那就没有什么事可干了，所以把 scm_cookie 结构“销毁”就完事了（52 行）。所谓把 scm_cookie 结构“销毁”，实际上是指把依附于这个结构的缓冲区释放。至于 scm_cookie 结构本身，是作为 sock_recvmsg( )的局部量存在于堆栈中，所以不存在释放空间的问题。这里要指出，只有在通过 recvmsg( )进入 sys_recvmsg( )时，才有可能在 msghdr 结构中为控制信息准备缓冲区，因为这个函数的参数之一是个 msghdr 结构指针；其他的函数都在程序中固定将 msghdr 结构内的指针 msg_control 设成 NULL。所以，只有通过 recvmsg( )才能接收到对方的身份消息，也只有通过 recvmsg( )才能接收对方传递过来的已打开文件的访问授权。

函数 put_cmsg( )将有关的附加信息（在这里是有关对方身份的信息）递交到用户空间去，其代码在 net/core/scm.c 中，有兴趣的读者可自行阅读。

如果在 scm_cookie 结构中包含有对已打开文件的访问授权，那就要把这些已打开文件（由发送进程打开）纳入到接收进程的已打开文件表中。相对来说，这就要复杂一点了，scm_detach_fds( )的代码也在 scm.c 中：

[sys_socketcall( ) > sys_recvfrom( ) > sock_recvmsg( ) > scm_recv( ) > scm_detach_fds( )]

```
203     void scm_detach_fds(struct msghdr *msg, struct scm_cookie *scm)
204     {
205         struct cmsghdr *cm = (struct cmsghdr*)msg->msg_control;
206
207         int fdmax = 0;
208         int fdnum = scm->fp->count;
209         struct file **fp = scm->fp->fp;
210         int *cmfptr;
211         int err = 0, i;
212
213         if (msg->msg_controllen > sizeof(struct cmsghdr))
214             fdmax = ((msg->msg_controllen - sizeof(struct cmsghdr))
215                 / sizeof(int));
216
217         if (fdnum < fdmax)
218             fdmax = fdnum;
219
220         for (i=0, cmfptr=(int*)CMSG_DATA(cm); i<fdmax; i++, cmfptr++)
221         {
222             int new_fd;
223             err = get_unused_fd( );
224             if (err < 0)
```

```
225                 break;
226             new_fd = err;
227             err = put_user(new_fd, cmfptr);
228             if (err) {
229                 put_unused_fd(new_fd);
230                 break;
231             }
232             /* Bump the usage count and install the file. */
233             get_file(fp[i]);
234             fd_install(new_fd, fp[i]);
235         }
236 
237         if (i > 0)
238         {
239             int cmlen = CMSG_LEN(i*sizeof(int));
240             if (!err)
241                 err = put_user(SOL_SOCKET, &cm->cmsg_level);
242             if (!err)
243                 err = put_user(SCM_RIGHTS, &cm->cmsg_type);
244             if (!err)
245                 err = put_user(cmlen, &cm->cmsg_len);
246             if (!err) {
247                 cmlen = CMSG_SPACE(i*sizeof(int));
248                 msg->msg_control += cmlen;
249                 msg->msg_controllen -= cmlen;
250             }
251         }
252         if (i < fdnum || (fdnum && fdmax <= 0))
253             msg->msg_flags |= MSG_CTRUNC;
254 
255         /*
256          * All of the files that fit in the message have had their
257          * usage counts incremented, so we just free the list.
258          */
259         __scm_destroy(scm);
260     }
```

代码中的 fdmax 表示接收方 msghdr 结构中用来接收这些已打开文件指针的缓冲区容量；而 fdnum 则是通过报文传递过来的已打开文件指针的个数，此项信息已经在一个临时的 scm_cookie 数据结构中。显然，这两个数值只能以较小者为准。读者在前面已经看到过，在 scm_cookie 结构中有个指针 fp，指向一个 scm_fp_list 结构，而这个结构中则有一个 file 结构指针数组，数组中的每个元素都是指针，指向发送进程的一个已打开文件的 file 结构。现在，发送方已经将这个指针传过来了，接收方要做的就是把它“安装”到它自己的已打开文件表中。这样做了以后，两个进程就可以共享这个已打开文件了。当然，共享的双方必须在同一台计算机上。

代码中从 220 行开始的 for 循环就是对传过来并且在缓冲区容量之内的每个指针做这件事。首先当然是在当前进程的打开文件表中找到一个空闲位置，其下标即是新的打开文件号 new_fd。然后，宏操

作 put_user( )把这个新的打开文件号写入到用户空间中准备好的缓冲区中（msg->msg_control 指向这块缓冲区）。前面我们讲到缓冲区的容量问题，就是指用户空间中的这块缓冲区是否足够于用来返回这些新的打开文件号。一旦把传过来的指针安装在接收进程的打开文件表中，相应的 file 结构就多了一个用户，所以要通过 get_file( )递增其共享计数。最后，就是把这个指针“安装”到接收进程（就是当前进程）的打开文件表中了，而 new_fd 则指明了在表中的位置。

前面讲过，msg->msg_control 指向用户空间的一块缓冲区，这块缓冲区实际上是由一个 cmsghdr 结构加上一个数据部分（用来返回新增的打开文件号）构成的。只要接收并安装了至少一个新的已打开文件，就要设置 cmsghdr 结构中的头部信息，包括实际使用的数据部分的长度 cmlen。最后，还要调用__scm_destroy( )，将依附于 scm_cookie 结构的动态分配的存储空间（在这里是 scm_fp_list 结构）释放掉。

进程间对已打开文件的这种访问授权，在某些应用中是很有意义的。以插口的使用为例，我们以前讲过，“有连接”插口的典型运用是：在 accept( )产生了一个新的已经建立起连接的插口以后，就 fork( )一个子进程，让子进程去为 client 方提供服务，而父进程本身则又进入 accept( )等待新的连接请求。不管怎么说，动态地 fork( )一个子进程的代价终究还是不小的。现在有了跨进程的访问授权，就可以预先创建若干个子进程，一旦 accept( )产生了一个新的插口，就可以把对该插口的访问权传给某个空闲的子进程，让它去提供服务。这样，就可以省去了因每次都 fork( )一个新进程而引起的延迟。

在上面这个情景中，我们假定在接收方的 receive_queue 队列中已经有报文在等待，所以接收进程不需要在 skb_recv_datagram( )中睡眠等待。如果接收队列中没有报文，那接收进程就需要睡眠等待，等到有报文到达时才被唤醒，这种情景读者在前面讲述 accept( )和 connect( )时已经看到过了。

再来看 Unix 域“无连接”插口的报文发送。

与 sock_recvmsg( )相对应的函数是 sock_sendmsg( )，其代码在文件 net/socket.c 中：

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( )]

```
501   int sock_sendmsg(struct socket *sock, struct msghdr *msg, int size)
502   {
503       int err;
504       struct scm_cookie scm;
505
506       err = scm_send(sock, msg, &scm);
507       if (err >= 0) {
508           err = sock->ops->sendmsg(sock, msg, size, &scm);
509           scm_destroy(&scm);
510       }
511       return err;
512   }
```

首先是对发送者身份以及附加控制信息的处理，inline 函数 scm_send( )的代码在 include/net/scm.h 中。与接收报文时相似，只有在通过 sendmsg( )进入 sys_sendmsg( )时才有可能随同报文发送这些附加信息，因为这个函数的参数之一是个 msghdr 结构指针，其他的函数都在程序中固定将 msghdr 结构内的指针 msg_control 设成 NULL。

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( )]

```
33      static __inline__ int scm_send(struct socket *sock, struct msghdr *msg,
34                                     struct scm_cookie *scm)
35      {
36          memset(scm, 0, sizeof(*scm));
37          scm->creds.uid = current->uid;
38          scm->creds.gid = current->gid;
39          scm->creds.pid = current->pid;
40          if (msg->msg_controllen <= 0)
41              return 0;
42          return __scm_send(sock, msg, scm);
43      }
```

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( ) > __scm_send( )]

```
114     int __scm_send(struct socket *sock, struct msghdr *msg, struct scm_cookie *p)
115     {
116         struct cmsghdr *cmsg;
117         int err;
118
119         for (cmsg = CMSG_FIRSTHDR(msg); cmsg; cmsg = CMSG_NXTHDR(msg, cmsg))
120         {
121             err = -EINVAL;
122
123             /* Verify that cmsg_len is at least sizeof(struct cmsghdr) */
124             /* The first check was omitted in <= 2.2.5. The reasoning was
125                that parser checks cmsg_len in any case, so that
126                additional check would be work duplication.
127                But if cmsg_level is not SOL_SOCKET, we do not check
128                for too short ancillary data object at all! Oops.
129                OK, let's add it...
130              */
131             if (cmsg->cmsg_len < sizeof(struct cmsghdr) ||
132                 (unsigned long)(((char*)cmsg - (char*)msg->msg_control)
133                         + cmsg->cmsg_len) > msg->msg_controllen)
134                 goto error;
135
136             if (cmsg->cmsg_level != SOL_SOCKET)
137                 continue;
138
139             switch (cmsg->cmsg_type)
140             {
141             case SCM_RIGHTS:
142                 err=scm_fp_copy(cmsg, &p->fp);
143                 if (err<0)
144                     goto error;
145                 break;
146             case SCM_CREDENTIALS:
147                 if (cmsg->cmsg_len != CMSG_LEN(sizeof(struct ucred)))
```

```
148                     goto error;
149                 memcpy(&p->creds, CMSG_DATA(cmsg), sizeof(struct ucred));
150                 err = scm_check_creds(&p->creds);
151                 if (err)
152                     goto error;
153                 break;
154             default:
155                 goto error;
156             }
157         }
158 
159         if (p->fp && !p->fp->count)
160         {
161             kfree(p->fp);
162             p->fp = NULL;
163         }
164         return 0;
165 
166     error:
167         scm_destroy(p);
168         return err;
169     }
```

读者可以对照前面的 scm_recv( )自行阅读这段代码。

把附加信息搭挂到报文上以后，就要通过具体的发送函数把报文发送出去了。对于 Unix 域的无连接插口，这个函数是 unix_dgram_sendmsg( )。这个函数比较大，我们分段阅读：

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( ) > unix_dgram_sendmsg( )]

```
1145    /*
1146     *  Send AF_UNIX data.
1147     */
1148 
1149    static int unix_dgram_sendmsg(struct socket *sock, struct msghdr *msg, int len,
1150                       struct scm_cookie *scm)
1151    {
1152        struct sock *sk = sock->sk;
1153        struct sockaddr_un *sunaddr=msg->msg_name;
1154        unix_socket *other = NULL;
1155        int namelen = 0; /* fake GCC */
1156        int err;
1157        unsigned hash;
1158        struct sk_buff *skb;
1159        long timeo;
1160 
1161        err = -EOPNOTSUPP;
1162        if (msg->msg_flags&MSG_OOB)
```

```
1163            goto out;
1164
1165        if (msg->msg_namelen) {
1166            err = unix_mkname(sunaddr, msg->msg_namelen, &hash);
1167            if (err < 0)
1168                goto out;
1169            namelen = err;
1170        } else {
1171            sunaddr = NULL;
1172            err = -ENOTCONN;
1173            other = unix_peer_get(sk);
1174            if (!other)
1175                goto out;
1176        }
1177
1178        if (sock->passcred && !sk->protinfo.af_unix.addr &&
1179            (err = unix_autobind(sock)) != 0)
1180            goto out;
1181
1182        err = -EMSGSIZE;
1183        if ((unsigned)len > sk->sndbuf - 32)
1184            goto out;
1185
```

首先，“无连接”插口不支持“编外”的 OOB 报文，也不支持除 MSG_DONTWAIT 和 MSG_NOSIGNAL 以外的任何标志位。如果在调用参数中提供了对方地址，那就先通过 unix_mkname( ) 将其“规格化”后备用。如果没有提供对方地址呢？那就是假定已经调用过 connect( ),设置好了通向目标插口的路径，所以通过 unix_peer_get( )就可以获得指向对方 sock 结构的指针。可是，要是实际上并没有调用过 connect( )呢？那当然不能再往前走了，所以此时的出错代码为－ENOTCONN。此外，如果插口的可选项 passcred 规定要随报文传递与插口身份有关的信息（注意与发送进程身份的区别），而又从未通过 bind( )为其指定一个地址，那就要通过 unix_autobind( )为其自动生成一个。插口的发送报文缓冲区大小记录在其 sock 结构的 sndbuf 字段中，但是要保留 32 字节用于控制目的，所以 1183 行据此检查报文的长度。这里要注意，虽然在报文发送之前和接收之后可以采用 iovec[ ]把报文分散存放在多个缓冲区中，但是在发送的过程中总是在同一个缓冲区中。这一来是对网络报文的模拟，二来也是因为通过 iovec[ ]提供的缓冲区是在用户空间，而不是在系统空间。我们继续往下看：

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( ) > unix_dgram_sendmsg( )]

```
1186        skb=sock_alloc_send_skb(sk,len,0,msg->msg_flags&MSG_DONTWAIT,&err);
1187        if (skb==NULL)
1188            goto out;
1189
1190        memcpy(UNIXCREDS(skb), &scm->creds, sizeof(struct ucred));
1191        if (scm->fp)
1192            unix_attach_fds(scm, skb);
```

```
1193
1194        skb->h.raw = skb->data;
1195        err = memcpy_fromiovec(skb_put(skb, len), msg->msg_iov, len);
1196        if (err)
1197            goto out_free;
1198
1199        timeo = sock_sndtimeo(sk, msg->msg_flags & MSG_DONTWAIT);
1200
```

接着，就是为报文的发送分配一个 sk_buff 结构，包括所需的缓冲区。函数 sock_sendmsg( )在通过插口 proto_ops 的结构调用 unix_dgram_sendmsg( )之前，先组装了一个临时的 scm_cookie 结构，其中的信息来自更上层的已组装、或者发送进程作为参数传递过来的 msghdr 结构中。现在载送报文的 sk_buff 结构和缓冲区已经分配好，所以把这些信息先拷贝进去（见 1190～1195 行）。这里的 unix_attach_fds( )处理对已打开文件的访问授权，前面已经讲到过了，其代码也在 af_unix.c 中：

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( ) > unix_dgram_sendmsg( ) > unix_attach_fds( )]

```
1135    static void unix_attach_fds(struct scm_cookie *scm, struct sk_buff *skb)
1136    {
1137        int i;
1138        for (i=scm->fp->count-1; i>=0; i--)
1139            unix_inflight(scm->fp->fp[i]);
1140        UNIXCB(skb).fp = scm->fp;
1141        skb->destructor = unix_destruct_fds;
1142        scm->fp = NULL;
1143    }
```

同样，如果所授权的已打开文件代表着 Unix 域插口，则要在传递过程中记下一笔账。读者可结合前面的 unix_notinflight( )阅读 unix_inflight( )的代码（net/unix/garbage.c）：

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( ) > unix_dgram_sendmsg( ) > unix_attach_fds( ) > unix_inflight( )]

```
116    /*
117     * Keep the number of times in flight count for the file
118     * descriptor if it is for an AF_UNIX socket.
119     */
120
121    void unix_inflight(struct file *fp)
122    {
123        unix_socket *s=unix_get_socket(fp);
124        if(s) {
125            atomic_inc(&s->protinfo.af_unix.inflight);
126            atomic_inc(&unix_tot_inflight);
127        }
128    }
```

然后，再将报文中的数据从用户空间的缓冲区中拷贝到依附于 sk_buff 结构的缓冲区中。根据 msghdr 结构中的具体设置，用户空间中的数据有可能分散在若干个缓冲区中，但是依附于 sk_buff 结构的缓冲区却只有一个，其大小应是用户空间中各个缓冲区中数据长度的总和。函数 memcpy_fromiovec( )与我们前面看到过的 memcpy_toiovec( )相似，只是拷贝的方向相反。

此外，inline 函数 skb_put( )，根据数据的总长度调整 sk_buff 结构中的指针 skb->tail 和当前数据长度 skb->len，而返回 skb->tail 的初值，也就是缓冲区的起点（1195 行），这段代码在 include/linux/skbuff.h 中：

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( ) > unix_dgram_sendmsg( ) > skb_put( )]

```
694    /**
695     *  skb_put - add data to a buffer
696     *  @skb: buffer to use
697     *  @len: amount of data to add
698     *
699     *  This function extends the used data area of the buffer. If this would
700     *  exceed the total buffer size the kernel will panic. A pointer to the
701     *  first byte of the extra data is returned.
702     */
703
704    static inline unsigned char *skb_put(struct sk buff *skb, unsigned int len)
705    {
706        unsigned char *tmp=skb->tail;
707        skb->tail+=len;
708        skb->len+=len;
709        if(skb->tail>skb->end) {
710            skb_over_panic(skb, len, current_text_addr( ));
711        }
712        return tmp;
713    }
```

最后，还要通过一个 inline 函数 sock_sndtimeo( )确定对发送过程的时间限制：

```
1249    static inline long sock_sndtimeo(struct sock *sk, int noblock)
1250    {
1251        return noblock ? 0 : sk->sndtimeo;
1252    }
```

所有这些准备工作都完成以后，就要把载送着报文的 sk_buff 结构送到目标插口一侧去了。我们继续往下看：

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( ) > unix_dgram_sendmsg( )]

```
1201    restart:
1202        if (!other) {
1203            err = -ECONNRESET;
```

```
1204            if (sunaddr == NULL)
1205                goto out_free;
1206
1207            other = unix_find_other(sunaddr, namelen, sk->type, hash, &err);
1208            if (other==NULL)
1209                goto out_free;
1210        }
1211
1212        unix_state_rlock(other);
1213        err = -EPERM;
1214        if (!unix_may_send(sk, other))
1215            goto out_unlock;
1216
1217        if (other->dead) {
1218            /*
1219             * Check with 1003.1g - what should
1220             * datagram error
1221             */
1222            unix_state_runlock(other);
1223            sock_put(other);
1224
1225            err = 0;
1226            unix_state_wlock(sk);
1227            if (unix_peer(sk) == other) {
1228                unix_peer(sk)=NULL;
1229                unix_state_wunlock(sk);
1230
1231                unix_dgram_disconnected(sk, other);
1232                sock_put(other);
1233                err = -ECONNREFUSED;
1234            } else {
1235                unix_state_wunlock(sk);
1236            }
1237
1238            other = NULL;
1239            if (err)
1240                goto out_free;
1241            goto restart;
1242        }
1243
1244        err = -EPIPE;
1245        if (other->shutdown&RCV_SHUTDOWN)
1246            goto out_unlock;
1247
```

如果插口在本次发送之前已经通过 connect( )建立了通向目标插口的路径，那么此时指针 other 已经指向对方的 sock 结构，否则即为 NULL。或者，虽然以前已经调用过 connect( )，但是在本次发送时

又给定了一个地址，此时 other 也是 0（见 1165 行）。所以，指针 other 为 NULL 时需要临时通过 unix_find_other( )找到对方的 sock 结构。这个函数我们在 connect( )一节中已经看到过了。找到了对方以后，还要通过 unix_may_send( )测试一下是否允许从我方插口向对方插口发送，测试的准则是对方并没有通过 connect( )把路径设置成通向"第三者"，unix_may_send( )的代码我们也已经在讲述"无连接"插口的 connect( )时看到过了。

虽然找到了对方插口，但是这个插口却并不一定还活着。我们在讲述"有连接"插口的 connect( )时已经讨论过这方面的问题，回过头去看一下会有助于理解 1217～1242 行的这一段代码。在寻找对方插口的 sock 结构的过程中，函数 unix_find_other( )和 unix_peer_get( )是二者必经其一的，而这二者都直接或间接地调用 sock_hold( )递增了对方插口的使用计数。所以 1223 行的 sock_put( )就是与这二者之一配对的，也就是将对方插口的使用计数递减，如果减后达到了 0 就将其 sock 结构所占的空间释放。如果本次发送的目标只是临时的，那么这就可以了，下面就转到 restart 处看看是否还能找到具有相同地址的其他插口（原来的插口撤销后可能又重新建立了），不过通常此时 unix_find_other( )会失败而出错返回。可是，如果这个目标是以前通过 connect( )设置的，那就又不一样了。首先是还要再调用一次 sock_put( )，因为当初在建立起通向目标的路径时也曾对目标插口的 sock 结构调用过一次 sock_hold( )，现在要把它抵消。其次，既然是原先通过 connect( )建立的比较稳固的伙伴关系（虽然是无连接模式），而现在对方已经"去世"，那就不像对临时给定的地址一样需要再找找有无相同地址的插口了，所以直接就将出错代码设置成－ECONNREFUSED，并转至 out_free 处返回。

现在只剩下最后一个可能的障碍了，那就是目标插口可能已经通过 shutdown( )将接收报文的功能关闭了（但是插口并未撤销）。注意在这种情况下返回的出错代码为－EPIPE。

至此，所有的关卡都已经通过了，下面就要将报文（即 sk_buff 结构）挂入目标插口的 sock 结构里的接收队列中。

再在 unix_dgram_sendmsg( )中继续往下看（af_unix.x）：

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( ) > unix_dgram_sendmsg( )]

```
1248		if (unix_peer(other) != sk &&
1249		    skb_queue_len(&other->receive_queue) > other->max_ack_backlog) {
1250		    if (!timeo) {
1251			err = -EAGAIN;
1252			goto out_unlock;
1253		    }
1254
1255		    timeo = unix_wait_for_peer(other, timeo);
1256
1257		    err = sock_intr_errno(timeo);
1258		    if (signal_pending(current))
1259			goto out_free;
1260
1261		    goto restart;
1262		}
1263
1264		skb_queue_tail(&other->receive_queue, skb);
1265		unix_state_runlock(other);
```

```
1266        other->data_ready(other, len);
1267        sock_put(other);
1268        return len;
1269
1270    out_unlock:
1271        unix_state_runlock(other);
1272    out_free:
1273        kfree_skb(skb);
1274    out:
1275        if (other)
1276            sock_put(other);
1277        return err;
1278    }
```

下面就是将一个“无连接”模式报文挂入接收队列的过程，这与“有连接”模式中将一个“连接请求”报文挂入 server 插口接收队列的过程很相似。所以，我们这里就不重复了，读者可以跟前面 sys_connect( )的代码对照阅读。

看完了“无连接”模式的报文接收与发送，我们再来看看“有连接”模式下的报文接收与发送。

与 unix_dgram_recvmsg( )相对应，Unix 域“有连接”模式的报文接收是通过 unix_stream_recvmsg( )完成的。这个函数的代码也在 af_unix.c 中：

[sys_socketcall( ) > sys_recvmsg( ) > unix_stream_recvmsg( )]

```
1149        static int unix_stream_recvmsg(struct socket *sock, struct msghdr *msg,
                           int size,
            . . . . . .
1500                       int flags, struct scm_cookie *scm)
1501    {
1502        struct sock *sk = sock->sk;
1503        struct sockaddr_un *sunaddr=msg->msg_name;
1504        int copied = 0;
1505        int check_creds = 0;
1506        int target;
1507        int err = 0;
1508        long timeo;
1509
1510        err = -EINVAL;
1511        if (sk->state != TCP_ESTABLISHED)
1512            goto out;
1513
1514        err = -EOPNOTSUPP;
1515        if (flags&MSG_OOB)
1516            goto out;
1517
1518        target = sock_rcvlowat(sk, flags&MSG_WAITALL, size);
1519        timeo = sock_rcvtimeo(sk, flags&MSG_DONTWAIT);
```

```
1520
1521        msg->msg_namelen = 0;
1522
```

以前讲过，“无连接”模式的插口是“无状态”的，而“有连接”模式的插口则是“有状态”的，其 sock 结构中的 state 就是用来实现一种“有限状态机”。所以这里要检查对方插口是否处于可以接收数据报文（而不是连接请求）的状态，实际上就是要检查连接是否已经建立好。

以前我们也讲过，“有连接”模式的通信支持所谓“编外”，即 OOB 报文的传递，这种报文在传递中具有较高的优先级，但是，那只是就一般而言，它实际上是为网络环境，特别是速度比较慢的网络环境而设计的。对 Unix 域的插口来说，收发双方都在同一台机器上,实际上并没有这种需要，所以 Unix 域的“有连接”插口并不支持这种报文。

调用参数中的 size 表示接收缓冲区的大小，也就是想要接收的字节数。如前所述，“有连接”模式的接收是跨越报文边界的。我们以前曾通过一个例子来说明，当接收队列中有两个报文，而第一个报文中的数据并没有将接收缓冲区填满时，就会继续从第二个报文中读取数据。可是，当时我们没有进一步说明，如果队列中只有一个报文，而又不能将接收缓冲区填满的话，那又当如何？这时候接收进程是继续等待新报文的到来，还是返回一个只是部分填满的缓冲区？现在就要回答这个问题了。这里在代码中有个局部量 target，就是表示对数据量的最低要求。如果接收队列中已经没有报文了，但是已经接收到缓冲区中的字节数还低于这个最低要求，那就要睡眠等待，否则就返回了（虽然不无遗憾）。通常，这个最低要求是 1。但是接收进程可以在参数 flags 中设置一个 MSG_WAITALL，表示“不达目的决不收兵”。所以在 1518 行中通过 sock_rcvlowat( )决定这个数值，然后设置当前的目标 target。

[sys_socketcall( ) > sys_recvmsg( ) > unix_stream_recvmsg( ) > sock_rcvlowat( )]

```
1254    static inline int sock_rcvlowat(struct sock *sk, int waitall, int len)
1255    {
1256        return (waitall ? len : min(sk->rcvlowat, len)) ? : 1;
1257    }
```

另一方面，也要通过 sock_rcvtimeo( )确定对发送过程的时间限制，这我们已经在前面看到过了。再继续往下看：

[sys_socketcall( ) > sys_recvmsg( ) > unix_stream_recvmsg( )]

```
1523        /* Lock the socket to prevent queue disordering
1524         * while sleeps in memcpy_tomsg
1525         */
1526
1527        down(&sk->protinfo.af_unix.readsem);
1528
1529        do
1530        {
1531            int chunk;
1532            struct sk_buff *skb;
1533
```

```
1534            skb=skb_dequeue(&sk->receive_queue);
1535            if (skb==NULL)
1536            {
1537                if (copied >= target)
1538                    break;
1539
1540                /*
1541                 *  POSIX 1003.1g mandates this order.
1542                 */
1543
1544                if ((err = sock_error(sk)) != 0)
1545                    break;
1546                if (sk->shutdown & RCV_SHUTDOWN)
1547                    break;
1548                err = -EAGAIN;
1549                if (!timeo)
1550                    break;
1551                up(&sk->protinfo.af_unix.readsem);
1552
1553                timeo = unix_stream_data_wait(sk, timeo);
1554
1555                if (signal_pending(current)) {
1556                    err = sock_intr_errno(timeo);
1557                    goto out;
1558                }
1559                down(&sk->protinfo.af_unix.readsem);
1560                continue;
1561            }
1562
1563            if (check_creds) {
1564                /* Never glue messages from different writers */
1565                if (memcmp(UNIXCREDS(skb), &scm->creds, sizeof(scm->creds)) != 0) {
1566                    skb_queue_head(&sk->receive_queue, skb);
1567                    break;
1568                }
1569            } else {
1570                /* Copy credentials */
1571                scm->creds = *UNIXCREDS(skb);
1572                check_creds = 1;
1573            }
1574
1575            /* Copy address just once */
1576            if (sunaddr)
1577            {
1578                unix_copy_addr(msg, skb->sk);
1579                sunaddr = NULL;
1580            }
1581
```

```
1582            chunk = min(skb->len, size);
1583            if (memcpy_toiovec(msg->msg_iov, skb->data, chunk)) {
1584                skb_queue_head(&sk->receive_queue, skb);
1585                if (copied == 0)
1586                    copied = -EFAULT;
1587                break;
1588            }
1589            copied += chunk;
1590            size -= chunk;
1591
1592            /* Mark read part of skb as used */
1593            if (!(flags & MSG_PEEK))
1594            {
1595                skb_pull(skb, chunk);
1596
1597                if (UNIXCB(skb).fp)
1598                    unix_detach_fds(scm, skb);
1599
1600                /* put the skb back if we didn't use it up.. */
1601                if (skb->len)
1602                {
1603                    skb_queue_head(&sk->receive_queue, skb);
1604                    break;
1605                }
1606
1607                kfree_skb(skb);
1608
1609                if (scm->fp)
1610                    break;
1611            }
1612            else
1613            {
1614                /* It is questionable, see note in unix_dgram_recvmsg.
1615                 */
1616                if (UNIXCB(skb).fp)
1617                    scm->fp = scm_fp_dup(UNIXCB(skb).fp);
1618
1619                /* put message back and return */
1620                skb_queue_head(&sk->receive_queue, skb);
1621                break;
1622            }
1623        } while (size);
1624
1625        up(&sk->protinfo.af_unix.readsem);
1626    out:
1627        return copied ? : err;
1628    }
```

读者已经看到，这里的主体是一个 do_while 循环。在循环中，每次从接收队列中摘下一个 sk_buff 结构，然后就从报文中读取数据和附加信息，直至缓冲区已经填满（size 变成了 0）；或者接收队列中已经没有报文，而接收的最低要求又已满足(见 1538 行)。如果缓冲区已经填满而报文中尚有数据剩余则将剩余部分退回接收队列（1603 行），以备下一次再来继续接收。反之，如果缓冲区未满而接收队列中已经没有报文，则视最低要求是否已经满足而决定是睡眠等待还是返回。这整个过程涉及队列操作，所以不允许受到打扰。同时，所需的时间又相对较长，不能单纯地靠加锁（spin_lock）来保护，所以需要通过内核信号量加以保护，这个内核信号量就在目标插口的 sock 结构中（见 1527、1551、1559 以及 1625 行）。

循环体中所调用的函数以及其他各种“要素”大部分都已经在前面出现过，所以我们基本上把这一段代码留给读者作为练习，只是对其中的几个函数及变量略作说明。

首先，inline 函数 skb_dequeue( )从给定队列（在这里是插口的接收队列）的前端摘下一个报文，即 sk_buff 数据结构（见 1534 行）。如果调用参数 flags 中的 MSG_PEEK 为 1，则在从报文中读取了数据和附加信息以后要通过 skb_queue_head( )将已经脱链的报文退还到接收队列中的前端（见 1620 行）。要注意的是，这种“偷看”以一个报文为限。也就是说，不管报文缓冲区多大，也不管最低要求的数值 target 多大，看了从队列中摘下的第一个报文并将它退回以后就结束了（见 1621 行）。这样的处理使程序得以简化（否则还得准备下一个临时的队列），道理上也讲得过去（因为是“偷看”嘛）。

另外，还有几个地方也要用到 skb_queue_head( )。一是当接收缓冲区已经填满而 sk_buff 中的数据尚有剩余时（1603 行），这里的 skb->len 为报文中剩余的数据量；二是当通过 memcpy_toiovec( )将报文中的数据拷贝到用户空间中的缓冲区时出了错（见 1583～1588 行）。例如，要是接收进程的页面映射表中缓冲区所在的页面为“写保护”，那当然就失败了，此时 memcpy_toiovec( )返回－EFAULT，而正常则返回 0。不过这并不一定意味着整个接收操作的失败，因为有可能在此之前已经成功地接收了一些数据。所以，此时要看变量 copied 的数值，它反映着已经将多少数据拷贝到了用户空间。还有一种情况，就是接收缓冲区尚未填满，所以就企图从队列中的下一个报文中再取一些数据，可是却发现这下一个报文中的身份信息与前一个中的不同，也就是说下一个报文是来自另一个发送进程（见 1563～1568 行）。在这种情况下，当然不应该将来自两个不同进程的报文“粘贴”在一起，所以也要把下一个报文退还到接收队列中。

读者也许感到奇怪，既然是“有连接”模式，那就只有通过已经建立了连接的插口才能发送报文，怎么又可能会有来自其他进程的报文呢？其实很简单，连接是建立在两个插口之间，而不是两个进程之间的。拥有已经建立好连接的插口的进程，既可以 fork( )出若干个子进程，也可以将对这个插口的访问通过 sendmsg( )授权给另一个进程。

接收了一个报文并读取了所载送的信息以后，要通过 kfree_skb( )释放报文的缓冲区(1607 行)，这个 inline 函数的代码在 include/linux/skbuff.h 中：

[sys_socketcall( ) > sys_recvmsg( ) > unix_stream_recvmsg( ) > kfree_skb( )]

```
209    /*
210     * If users==1, we are the only owner and are can avoid redundant
211     * atomic change.
212     */
213
214    /**
```

```
215      * kfree skb - free an sk_buff
216      * @skb: buffer to free
217      *
218      * Drop a reference to the buffer and free it if the usage count has
219      * hit zero.
220      */
221
222     static inline void kfree_skb(struct sk_buff *skb)
223     {
224     if (atomic_read(&skb->users) == 1 || atomic_dec_and_test(&skb->users))
225             __kfree_skb(skb);
226     }
```

这里__kfree_skb( )的代码在 net/core/skbuff.c 中：

[sys_socketcall( ) > sys_recvmsg( ) > unix_stream_recvmsg( ) > kfree_skb( ) > __kfree_skb( )]

```
272     void __kfree_skb(struct sk_buff *skb)
273     {
274         if (skb->list) {
275             printk(KERN_WARNING "Warning: kfree_skb passed an skb still "
276                    "on a list (from %p).\n", NET_CALLER(skb));
277             BUG( );
278         }
279
280         dst_release(skb->dst);
281         if(skb->destructor) {
282             if (in_irq( )) {
283                 printk(KERN_WARNING "Warning: kfree_skb on hard IRQ %p\n",
284                     NET_CALLER(skb));
285             }
286             skb->destructor(skb);
287         }
288     #ifdef CONFIG_NETFILTER
289         nf_conntrack_put(skb->nfct);
290     #endif
291         skb_headerinit(skb, NULL, 0);  /* clean state */
292         kfree_skbmem(skb);
293     }
```

这个函数也许比读者想像的要复杂。我们把这里调用的一些函数留给读者，只是要提请读者注意：如果 sk_buff 结构中的函数指针 destructor 为非 0，就要先调用这个函数。那么，具体到 unix_stream_recvmsg( )，这个指针是什么呢？后面读者就会看到，这是 sock_wfree( )，是由发送报文的一方安排好了的。

最后，就是“有连接”模式的报文发送了，这是由 af_unix.c 中的函数 unix_stream_sendmsg( )完成的。

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( ) > unix_stream_sendmsg( )]

```
1281    static int unix_stream_sendmsg(struct socket *sock, struct msghdr *msg, int len,
1282                    struct scm_cookie *scm)
1283    {
1284        struct sock *sk = sock->sk;
1285        unix_socket *other = NULL;
1286        struct sockaddr_un *sunaddr=msg->msg_name;
1287        int err,size;
1288        struct sk_buff *skb;
1289        int limit=0;
1290        int sent=0;
1291
1292        err = -EOPNOTSUPP;
1293        if (msg->msg_flags&MSG_OOB)
1294            goto out_err;
1295
1296        if (msg->msg_namelen) {
1297            err = (sk->state==TCP_ESTABLISHED ? -EISCONN : -EOPNOTSUPP);
1298            goto out_err;
1299        } else {
1300            sunaddr = NULL;
1301            err = -ENOTCONN;
1302            other = unix_peer_get(sk);
1303            if (!other)
1304                goto out_err;
1305        }
1306
1307        if (sk->shutdown&SEND_SHUTDOWN)
1308            goto pipe_err;
1309
```

如前所述，在 Unix 域中虽然是“有连接”模式也不支持 OOB 报文。并且除 MSG_DONTWAIT 和 MSG_NOSIGNAL 以外所有的标志位都不允许使用。另一方面，既然是“有连接”模式的发送，就不允许指定接收方地址。通过 unix_peer_get( )取得对方插口的 sock 结构指针 other 以后，这里并没有检验它是否还活着，以及是否已经关闭了报文接收。这是为什么呢？下面读者就会看到，整个发送过程是通过一个循环来完成的，对这两个条件的检查在每次循环中都要进行，而不是只检查一次。不过，倒是要先检查一下发送方插口本身是否已经关闭报文发送（见 1307 行）。再往下看：

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( ) > unix_stream_sendmsg( )]

```
1310        while(sent < len)
1311        {
1312            /*
1313             * Optimisation for the fact that under 0.01% of X messages typically
1314             * need breaking up.
```

```
1315             */
1316
1317            size=len-sent;
1318
1319            /* Keep two messages in the pipe so it schedules better */
1320            if (size > sk->sndbuf/2 - 16)
1321                size = sk->sndbuf/2 - 16;
1322
1323            /*
1324             *  Keep to page sized kmalloc( )'s as various people
1325             *  have suggested. Big mallocs stress the vm too
1326             *  much.
1327             */
1328
1329            if (size > PAGE_SIZE-16)
1330            limit = PAGE_SIZE-16; /* Fall back to a page if we can't
                                grab a big buffer this instant */
1331            else
1332                limit = 0;  /* Otherwise just grab and wait */
1333
1334            /*
1335             *  Grab a buffer
1336             */
1337             a
1338             skb = sock_alloc_send_skb(sk,size,limit,
                                    msg->msg_flags&MSG_DONTWAIT, &err);
1339
1340            if (skb==NULL)
1341                goto out_err;
1342
1343            /*
1344             *  If you pass two values to the sock_alloc_send_skb
1345             *  it tries to grab the large buffer with GFP_BUFFER
1346             *  (which can fail easily), and if it fails grab the
1347             *  fallback size buffer which is under a page and will
1348             *  succeed. [Alan]
1349             */
1350            size = min(size, skb_tailroom(skb));
1351
1352            memcpy(UNIXCREDS(skb), &scm->creds, sizeof(struct ucred));
1353            if (scm->fp)
1354                unix_attach_fds(scm, skb);
1355
1356    if ((err = memcpy_fromiovec(skb_put(skb,size), msg->msg_iov, size)) != 0) {
1357                kfree_skb(skb);
1358                goto out_err;
1359            }
1360
```

```
1361            unix_state_rlock(other);
1362
1363            if (other->dead || (other->shutdown & RCV_SHUTDOWN))
1364                goto pipe_err_free;
1365
1366            skb_queue_tail(&other->receive_queue, skb);
1367            unix_state_runlock(other);
1368            other->data_ready(other, size);
1369            sent+=size;
1370        }
1371        sock_put(other);
1372        return sent;
1373
1374    pipe_err_free:
1375        unix_state_runlock(other);
1376        kfree_skb(skb);
1377    pipe_err:
1378        if (sent==0 && !(msg->msg_flags&MSG_NOSIGNAL))
1379            send_sig(SIGPIPE,current,0);
1380        err = -EPIPE;
1381    out_err:
1382            if (other)
1383            sock_put(other);
1384        return sent ? : err;
1385    }
```

为什么“有连接”模式的报文发送要通过一个循环来完成，而在“无连接”模式中就不需要循环呢？这又跟二者的语义有关。

“无连接”模式是面向报文的，必须维持报文的边界。用户（进程）交下来一块数据，不管多大（只要不超过极限）也要在一个报文中发送出去。至于用户交下来的数据块大小是否合理，那是用户的事。如果用户交下来的数据块太大，则立即失败返回。

而“有连接”模式就不同了。“有连接”模式是面向字节流的，所以可以在发送方将一个大报文分割成若干较小的报文来发送。这样比较有利于改善报文传递的效率，也减轻了用户的负担。那么，如果用户交下的报文太大，把它分割成多大才是合适的呢？或者说，多大的报文才需要分割呢？代码中把这个界线定为 sk->sndbuf 的一半（16 字节保留用于头部结构，在下面的讨论中我们都把它略去不计）。插口的 sock 结构中有个参数 sndbuf，是可以通过 setsockopt( )设置和改变的。这个参数大致上决定了插口在正常条件下可以占用作为发送缓冲区的总的存储区大小。那么，为什么是这个值的一半呢？答案是，把它分成两半来用有利于提高效率。这样，就可以使得当发送方在准备后一半时，接收方已经在接收前一半了。通过对两个缓冲区的循环使用和流通，就可以形成一个“流水线”，从而提高效率。在网络环境下或者发送方和接收方在两个不同处理器上运行时，这种效果就尤为突出。可是，有时候由于系统中资源使用和周转的限制，这个值的一半也还是太大而一时分配不到所需的缓冲区，这时候怎么办呢？当然可以睡眠等待，但是更好的办法是退而求其次，分配一块再小一些的缓冲区，这样显然更有利于资源的周转和在这种“逆境”下效率的提高。不过，小到一个页面以下就不合适了，因为以页面为单位分配空间比较简单，效率也较高。这就是代码第 1330 行将变量 limit 设置成（PAGE_SIZE

—16）的道理。这个变量的作用，要与 sock_alloc_send_skb( )的代码结合起来看才清楚。这个函数的代码在 net/core/sock.c 中：

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( ) > unix_stream_sendmsg( ) > sock_alloc_send_skb( )]

```
741     /*
742      *  Generic send/receive buffer handlers
743      */
744
745     struct sk_buff *sock_alloc_send_skb(struct sock *sk, unsigned long size,
746                 unsigned long fallback, int noblock, int *errcode)
747     {
748         int err;
749         struct sk_buff *skb;
750         long timeo;
751
752         timeo = sock_sndtimeo(sk, noblock);
753
754         while (1) {
755             unsigned long try_size = size;
756
757             err = sock_error(sk);
758             if (err != 0)
759                 goto failure;
760
761             /*
762              *  We should send SIGPIPE in these cases according to
763              *  1003.1g draft 6.4. If we (the user) did a shutdown( )
764              *  call however we should not.
765              *
766              *  Note: This routine isnt just used for datagrams and
767              *  anyway some datagram protocols have a notion of
768              *  close down.
769              */
770
771             err = -EPIPE;
772             if (sk->shutdown&SEND_SHUTDOWN)
773                 goto failure;
774
775             if (atomic_read(&sk->wmem_alloc) < sk->sndbuf) {
776                 if (fallback) {
777                     /* The buffer get won't block, or use the atomic queue.
778                     * It does produce annoying no free page messages still.
779                     */
780                     skb = alloc_skb(size, GFP_BUFFER);
781                     if (skb)
782                         break;
```

```
783                     try_size = fallback;
784                 }
785                 skb = alloc_skb(try_size, sk->allocation);
786                 if (skb)
787                     break;
788                 err = -ENOBUFS;
789                 goto failure;
790             }
791 
792             /*
793              *  This means we have too many buffers for this socket already.
794              */
795 
796             set_bit(SOCK_ASYNC_NOSPACE, &sk->socket->flags);
797             set_bit(SOCK_NOSPACE, &sk->socket->flags);
798             err = -EAGAIN;
799             if (!timeo)
800                 goto failure;
801             if (signal_pending(current))
802                 goto interrupted;
803             timeo = sock_wait_for_wmem(sk, timeo);
804         }
805 
806         skb_set_owner_w(skb, sk);
807         return skb;
808 
809     interrupted:
810         err = sock_intr_errno(timeo);
811     failure:
812         *errcode = err;
813         return NULL;
814     }
```

这个函数先试着按 size 大小分配空间（注意 755 行将 try_size 设置成 size），分配时的优先级别为 GFP_BUFFER。如果行，就万事大吉了。不行的话，就要看参数 fallback。要是 fallback 为 0，那就或者通过 sock_wait_for_wmem( )睡眠等待，或者失败返回，具体取决于参数 noblock。可是，要是 fallback 非 0 的话，那就再试着按 fallback 的大小再分配一次，这一次分配的优先级别则改成 sk->allocation，通常是比 GFP_BUFFER 高一些（事实上，在 sock_init_data( )中将这个优先级设置成 GFP_KERNEL），而 fallback 则通常比 size 要小。这样，第二次分配成功的希望就相当大了。当然，还有可能再失败，那就要睡眠等待或失败返回了。换言之，当 fallback 为 0 时的空间分配是硬性的，成就成，不成就不成；而 fallback 为非 0 时的空间分配则是软性的，能按 size 的大小分配最好，不成就退而求其次，按 fallback 的大小进行分配。具体的缓冲区分配由 sock_wmalloc( )进行，其代码也在 sock.c 中：

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( ) > unix_stream_sendmsg( ) > sock_alloc_send_skb( ) > sock_wmalloc( )]

```
654     /*
655      * Allocate a skb from the socket's send buffer.
656      */
657      struct sk_buff *sock_wmalloc(struct sock *sk, unsigned long size,
                                int force, int priority)
658     {
659         if (force || atomic_read(&sk->wmem_alloc) < sk->sndbuf) {
660             struct sk_buff * skb = alloc_skb(size, priority);
661             if (skb) {
662                 skb_set_owner_w(skb, sk);
663                 return skb;
664             }
665         }
666         return NULL;
667     }
```

插口的 sock 结构中有个整数 wmem_alloc，记录着插口正占用着的缓冲区字节数。当分配缓冲区成功时就由这里的 skb_set_owner_w( )将分配得的实际大小加到 wmem_alloc 中，而释放时则从中减去。这个函数虽然小，却起着很重要的作用，其代码在 include/net/sock.h 中：

[sys_socketcall( ) > sys_sendmsg( ) > sock_sendmsg( ) > unix_stream_sendmsg( ) > sock_alloc_send_skb( ) > sock_wmalloc( ) > skb_set_owner_w( )]

```
1125    /*
1126     *  Queue a received datagram if it will fit. Stream and sequenced
1127     *  protocols can't normally use this as they need to fit buffers in
1128     *  and play with them.
1129     *
1130     *  Inlined as it's very short and called for pretty much every
1131     *  packet ever received.
1132     */
1133
1134    static inline void skb_set_owner_w(struct sk_buff *skb, struct sock *sk)
1135    {
1136        sock_hold(sk);
1137        skb->sk = sk;
1138        skb->destructor = sock_wfree;
1139        atomic_add(skb->truesize, &sk->wmem_alloc);
1140    }
```

注意这里将 sk_buff 结构中的函数指针 destructor 设置成指向 sock_wfree( )。这样，报文的接收方在释放这个数据结构时就会先调用这个函数。

回到 unix_stream_sendmsg( )的代码中的第 1338 行，这里的调用参数 limit 就是上面的 fallback。所以，当 size 小于一个页面时 limit 为 0，此时对缓冲区分配的要求是硬性的，因为不能比一个页面再小了。可是，当 size 大于一个页面时，则 limit 的大小为一个页面，此时就可以“讨价还价”了。读者不妨回过去看一下 unix_dgram_sendmsg( )的代码，那里对缓冲区分配的要求是硬性的，这是因为“无连

接”模式的报文发送不允许改变报文的边界。通过 sock_alloc_send_skb( )分配了缓冲区以后，缓冲区的实际大小可以调用 skb_tailroom( )来获取。

从这里往下的代码，读者应该已经很熟悉了。不过，不知读者看了以后是否感觉到有些异样？我们在 unix_stream_connect( )中和 unix_dgram_sendmsg( )中都曾看到，在将报文挂入接收队列之前要测试队列长度；如果其长度达到了某个限额就要睡眠等待。可是，在这里却没有看到。这是为什么？难道“有连接”模式的报文发送就不存在这个问题吗？是的，Unix 域“有连接”模式的报文发送确实不存在使对方的接收队列过长的问题。我们知道，连接是建立于插口之间，而不是进程之间。同时，我们已经看到，一个插口可以用作发送缓冲区的空间大小是有限制的。当发送方插口将缓冲区用光（取决于其 sock 结构中的 sndbuf），也即全部挂入到接收方插口的接收队列中去以后，就再也分配不到缓冲区，而只好在 sock_alloc_send_skb( )中睡眠等待了。等待什么呢？等待接收方进程从队列中接收报文而将缓冲区释放出来。那时候，接收方就会在释放报文的 sk_buff 结构是通过其函数指针调用 sock_wfree( )，其代码在 net/core/sock.c 中：

[sys_socketcall( ) > sys_recvmsg( ) > unix_stream_recvmsg( ) > kfree_skb( ) > __kfree_skb( ) > sock_wfree( )]

```
631    /*
632     * Write buffer destructor automatically called from kfree_skb.
633     */
634    void sock_wfree(struct sk_buff *skb)
635    {
636        struct sock *sk = skb->sk;
637
638        /* In case it might be waiting for more memory. */
639        atomic_sub(skb->truesize, &sk->wmem_alloc);
640        sk->write_space(sk);
641        sock_put(sk);
642    }
```

这里调整了 sk->wmem_alloc 的数值，因为发送方插口实际占用的缓冲区减少了。而 sock 结构中的另一个函数指针 write_space 则在创建插口时（见 unix_create1( )的代码，482 行）设置成指向 unix_write_space( )，这个函数将可能因分配不到报文缓冲区而将正在睡眠等待的发送方进程唤醒，其代码在 net/unix/af_unix.x 中：

[sys_socketcall( ) > sys_recvmsg( ) > unix_stream_recvmsg( ) > kfree_skb( ) > __kfree_skb( ) > sock_wfree( ) > unix_write_space( )]

```
299    static void unix_write_space(struct sock *sk)
300    {
301        read_lock(&sk->callback_lock);
302        if (unix_writable(sk)) {
303            if (sk->sleep && waitqueue_active(sk->sleep))
304                wake_up_interruptible(sk->sleep);
305            sk_wake_async(sk, 2, POLL_OUT);
```

```
306         }
307         read_unlock(&sk->callback_lock);
308     }
```

当睡眠中的发送方进程被唤醒，再次试图分配报文缓冲区时，就能成功了。

最后还要强调指出，发送方/接收方的关系与 client/server 两方的关系完全是两码事。前者是对一个具体的报文而言的，而后者仅与连接的建立有关，一旦建立了连接以后，任何一方都可以是一个报文的发送方。

7.8 插口的关闭

插口的关闭，即撤销，是通过文件系统界面的 close 系统调用完成的。读者不妨回想一下前面讲过的数据结构之间的联系，看看从插口的打开文件号和当前进程的 task_struct 结构开始，怎样才能找到插口的文件操作跳转表（数据结构 socket_file_ops）。该结构中的函数指针 release 指向 sock_close( )，其代码在 net/socket.c 中：

[sys_close( ) > filp_close( ) > fput( ) > sock_close( )]

```
692     int sock_close(struct inode *inode, struct file *filp)
693     {
694         /*
695          *  It was possible the inode is NULL we were
696          *  closing an unfinished socket.
697          */
698
699         if (!inode)
700         {
701             printk(KERN_DEBUG "sock_close: NULL inode\n");
702             return 0;
703         }
704         sock_fasync(-1, filp, 0);
705         sock_release(socki_lookup(inode));
706         return 0;
707     }
```

首先是对插口的 fasync_list 的处理。我们在讲述 connect( )时讲到过，server 方进程可以异步地等待来自 client 插口的连接请求。实际上，连接请求只是一个特例，异步接收同样也适用于普通报文的接收。在 socket 结构中有个 fasync_list 队列，如果想要在一个 server 插口上异步等待连接请求，或者从一个插口异步地接收报文，就可以通过 ioctl( )系统调用分配一个 fasync_struct 数据结构,并将其挂入插口的 fasync_list 队列。现在插口既然要撤销了，当然要把 fasync_list 队列中的 fasync_struct 结构全都释放掉。函数 sock_fasync( )的代码也在 socket.c 中，由于代码比较简单，我们就不详述了。接着，看 sock_release( )，其代码也在 socket.c 中：

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( )]

```
476	/**
477	 *	sock_release	-	close a socket
478	 *	@sock: socket to close
479	 *
480	 *	The socket is released from the protocol stack if it has a release
481	 *	callback, and the inode is then released if the socket is bound to
482	 *	an inode not a file.
483	 */
484
485	void sock_release(struct socket *sock)
486	{
487		if (sock->ops)
488			sock->ops->release(sock);
489
490		if (sock->fasync_list)
491			printk(KERN_ERR "sock_release: fasync list not empty!\n");
492
493		sockets_in_use[smp_processor_id( )].counter--;
494		if (!sock->file) {
495			iput(sock->inode);
496			return;
497		}
498		sock->file=NULL;
499	}
```

用于 Unix 域插口的两个 proto_ops 结构，即 unix_stream_ops 和 unix_dgram_ops，其中的 release 指针都指向 unix_release( )，其代码在 af_unix.c 中给出：

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( ) > unix_release( )]

```
525	static int unix_release(struct socket *sock)
526	{
527		unix_socket *sk = sock->sk;
528
529		if (!sk)
530			return 0;
531
532		sock->sk = NULL;
533
534		return unix_release_sock (sk, 0);
535	}
```

函数 unix_release_sock( )的代码也在同一文件 af_unix.c 中，我们分两段来看：

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( ) > unix_release( )

> unix_release_sock( )]

```
353	static int unix_release_sock (unix_socket *sk, int embrion)
354	{
355		struct dentry *dentry;
356		struct vfsmount *mnt;
357		unix_socket *skpair;
358		struct sk_buff *skb;
359		int state;
360	
361		unix_remove_socket(sk);
362	
363		/* Clear state */
364		unix_state_wlock(sk);
365		sock_orphan(sk);
366		sk->shutdown = SHUTDOWN_MASK;
367		dentry = sk->protinfo.af_unix.dentry;
368		sk->protinfo.af_unix.dentry=NULL;
369		mnt = sk->protinfo.af_unix.mnt;
370		sk->protinfo.af_unix.mnt=NULL;
371		state = sk->state;
372		sk->state = TCP_CLOSE;
373		unix_state_wunlock(sk);
374	
375		wake_up_interruptible_all(&sk->protinfo.af_unix.peer_wait);
376	
377		skpair=unix_peer(sk);
378	
379		if (skpair!=NULL) {
380			if (sk->type==SOCK_STREAM) {
381				unix_state_wlock(skpair);
382				skpair->shutdown=SHUTDOWN_MASK; /* No more writes*/
383				if (!skb_queue_empty(&sk->receive_queue) || embrion)
384					skpair->err = ECONNRESET;
385				unix_state_wunlock(skpair);
386				skpair->state_change(skpair);
387				read_lock(&skpair->callback_lock);
388				sk_wake_async(skpair, 1, POLL_HUP);
389				read_unlock(&skpair->callback_lock);
390			}
391			sock_put(skpair); /* It may now die */
392			unix_peer(sk) = NULL;
393		}
394	
```

以前我们讲过，内核中有个杂凑表 unix_socket_table[ ]，每个插口的 sock 结构都挂在杂凑表中的某一个队列中，而 Unix 域 sock 结构中的 protoinfo.af_unix.list 则指向该队列的头部。现在，第一件事就

是通过 unix_remove_socket( )将插口的 sock 结构从杂凑表的队列中脱链，其代码在 net/unix/af_unix.c 中：

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( ) > unix_release( ) > unix_release_sock( ) > unix_remove_socket( )]

```
233     static __inline__ void unix_remove_socket(unix_socket *sk)
234     {
235         write_lock(&unix_table_lock);
236         __unix_remove_socket(sk);
237         write_unlock(&unix_table_lock);
238     }
```

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( ) > unix_release( ) > nix_release_sock( ) > unix_remove_socket( ) > __unix_remove_socket( )]

```
203     static void __unix_remove_socket(unix_socket *sk)
204     {
205         unix_socket **list = sk->protinfo.af_unix.list;
206         if (list) {
207             if (sk->next)
208                 sk->next->prev = sk->prev;
209             if (sk->prev)
210                 sk->prev->next = sk->next;
211             if (*list == sk)
212                 *list = sk->next;
213             sk->protinfo.af_unix.list = NULL;
214             sk->prev = NULL;
215             sk->next = NULL;
216             __sock_put(sk);
217         }
218     }
```

下面，就要通过 sock_orphan( )改变 sock 结构中的一些状态信息以及一些指针了（sock.h)。

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( ) > unix_release( ) > nix_release_sock( ) > unix_remove_socket( )]

```
998     /* Detach socket from process context.
999      * Announce socket dead, detach it from wait queue and inode.
1000     * Note that parent inode held reference count on this struct sock,
1001     * we do not release it in this function, because protocol
1002     * probably wants some additional cleanups or even continuing
1003     * to work with this socket (TCP).
1004     */
1005    static inline void sock_orphan(struct sock *sk)
1006    {
```

```
1007        write_lock_bh(&sk->callback_lock);
1008        sk->dead = 1;
1009        sk->socket = NULL;
1010        sk->sleep = NULL;
1011        write_unlock_bh(&sk->callback_lock);
1012    }
```

首先就是把 dead 置成 1，表示这个插口已经死亡了，我们在前面的一些函数中曾多次看到对这个状态信息的检测。接着，就把 sock 结构中指向相应 socket 结构的指针设成了 NULL。这里 write_lock_bh( ) 是个宏操作，定义于 include/linux/spinlock.h 中：

```
20    #define write_lock_bh(lock)      \
              do { local_bh_disable( ); write_lock(lock); } while (0)
```

从代码中可以看到，这里加了两把锁。第一把锁是在上一层的 unix_release_sock( )中通过 unix_state_wlock( )加上的，此函数的定义在文件 af_unix.c 中给出：

```
39    #define unix_state_wlock(s) write_lock(&(s)->protinfo.af_unix.lock)
```

每一把“锁”，在锁上时都可以有两个状态，即“读”和“写”。这里的锁是 sock 结构中的 protoinfo.af_unix.lock。注意，write_lock( )并不是“锁上防写”，而是“为写而锁”。凡是要改变 sock 结构中任何内容时，都要将这把锁锁上至“写”状态。同时，哪怕只是要读 sock 结构中的内容，也得要加锁，不是锁至“写”状态就是锁至“读”状态。这样，只要有一个进程在写，其他的进程（在不同的处理器上运行，下同）就都不能读（当然也不能写），而只能在 write_lock( )或 read_lock( )中等待解锁（并不睡眠!）。反之，只要至少有一个进程在读，其他的进程就不能写，而只好在 write_lock( )中等待解锁，但是却可以顺利通过 read_lock( )。只有这样，才能保证任何进程都不会在改变内容的中途从中读出。可是，改变 sock 结构内容的过程往往需要比较长的时间，而仅仅要读 sk->dead 的内容就没有必要等所有的改变都完成，所以就为这个目的专门设置了另外一把锁，那就是 sock 结构中的 callback_lock。凡是需要读 sk->dead 的内容时，就至少要锁上其中的一把，但是哪一把都可以，即“写”状态或“读”状态都可以。

完成了对 sock 结构的改变以后，就可以唤醒正在睡眠中等待着要使用该插口的进程了。这样的进程有两种，一种是等待着报文的到达，另一种则是等待着将报文挂入本插口的接收队列中。这些进程被唤醒以后都会再一次检测 sk->dead，当发现这个插口已经死亡时就会出错返回。

在讲述 connect( )的时候，读者已经看到在 sock 结构中有个指针 pair，而在 af_unix.c 中对函数 unix_peer( )的定义就是：

```
139    #define unix_peer(sk) ((sk)->pair)
```

已经建立起连接的两个“有连接”模式插口互相通过这个指针指向对方的 sock 结构。至于“无连接”模式的插口，则为之调用了 connect( )的插口通过这个指针单向地指向对方的 sock 结构。同时，只要一个插口的这个指针非 0，则它就是对方插口的一个用户，而对方插口的用户计数就包含了这个插口，所以要通过 sock_put( )递减对方的用户计数（见 391 行），并且将这个指针清 0。由于“有连接”模式插口的“连接”是双向的，此时还要对连接的对方也作一些处理（380～390 行）。首先要切断对方继续

向我方发送报文的功能。其次，如果我方的接收队列中还有报文（在 unix_release( )中调用 unix_release_sock( )时的调用参数 embrion 为 0），就要告知对方，把对方 sock 结构中的出错代码 err 设成 ECONNRESET。这样，如果有进程正在系统调用中对其进行某些操作，就会出错返回，否则下一次为对方插口而进入系统调用时也会出错返回。内核代码中有个 inline 函数 sock_error( )，定义于 include/net/sock.h 中：

```
1191    /*
1192     * Recover an error report and clear atomically
1193     */
1194
1195    static inline int sock_error(struct sock *sk)
1196    {
1197        int err=xchg(&sk->err,0);
1198        return -err;
1199    }
```

这个 inline 函数一方面将 sk->err 的内容读入变量 err 中，一方面将 sk->err 清 0。在插口操作的一些关键函数和关键路径上都安排了对 sock 结构中的这个出错代码的检测。例如，在 sock_alloc_send_skb( )的 while 循环中以及在 unix_stream_recvmsg( )的 do_while 循环中都安排了这样的检测。

此外，sock 结构中还有个函数指针 state_change，在 sock 结构初始化时设置成指向 sock_def_wakeup( )。每当一个插口的状态改变时，就要通过这个指针调用相应的函数，以唤醒可能正在睡眠等待该插口改变状态的进程。现在，一对互相连接的插口中有一个已经关闭了，这当然引起另一方的状态改变，所以要调用这个函数（386 行）。

最后，还要通过 sk_wake_async( )向所有正在对方插口上异步操作的进程发出 SIGIO 信号，让它们在对方插口上进行一次操作，从而得知该插口的状态改变。

继续往下看函数 unix_release_sock( )代码的余下部分：

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( ) > unix_release( ) > unix_release_sock( )]

```
395         /* Try to flush out this socket. Throw out buffers at least */
396
397         while((skb=skb_dequeue(&sk->receive queue))!=NULL)
398         {
399             if (state==TCP_LISTEN)
400                 unix_release_sock(skb->sk, 1);
401             /* passed fds are erased in the kfree_skb hook        */
402             kfree_skb(skb);
403         }
404
405         if (dentry) {
406             dput(dentry);
407             mntput(mnt);
```

```
408         }
409
410         sock_put(sk);
411
412         /* ---- Socket is dead now and most probably destroyed ---- */
413
414         /*
415          * Fixme: BSD difference: In BSD all sockets connected to use get
416          *     ECONNRESET and we die on the spot. In Linux we behave
417          *     like files and pipes do and wait for the last
418          *     dereference.
419          *
420          * Can't we simply set sock->err?
421          *
422          *     What the above comment does talk about? --ANK(980817)
423          */
424
425         if (atomic_read(&unix_tot_inflight))
426             unix_gc();      /* Garbage collect fds */
427
428         return 0;
429     }
```

如果插口的接收队列中有报文，当然要逐个把它们摘下来释放。读者已经在前一节中看到过函数kfree_skb( )的代码。但是当插口的状态为TCP_LISTEN时是个特殊情况，因为此时接收队列中的报文都是连接请求。对方已经为server插口创建好一个sock结构，就等着server方进程通过accept( )为server插口创建一个新的socket结构，并使它们互相挂上钩。现在server插口既然撤销了，就要把所有这些由client方创建好的sock结构也撤销，所以这里对其递归调用unix_release_sock( )。

我们知道，通过bind( )为插口指定一个常规的、文件名形式的“地址”时，要在文件系统的目录树中创建一个节点，并使sock结构中的指针protinfo.af_unix.dentry指向内存中代表着这个目录项的dentry结构，同时也递增了这个结构中的使用计数。现在，如果这个dentry结构存在（见405行），就要通过dput( )递减它的使用计数，要是递减到了0就要将数据结构释放。对目录项所在文件系统的安装“连接件”也是一样，要通过mntput( )递减其使用计划数，要是递减到了0就要将此文件系统拆除。

对sock结构本身也要递减其使用计数。同样，如果递减到了0就要将此结构释放。如果递减后不是0呢？那就说明有别的进程还在使用这个结构。这是可能的，毕竟对sock结构所加的锁在373行处就已解除了。但是，不管是谁，对此结构调用的最后一次sock_put( )总会将其释放。

最后，还有个重要的事情要做。以前讲过，可以通过sendmsg( )把对已打开文件的访问授权给其他进程。发送报文的一方要在unix_attach_fds( )中通过unix_inflight( )将这些已打开文件记下账，表示对这些已打开文件的访问授权已经在发送的过程中，即所谓“inflight”（在飞行中）。而接收方则在接收到这些授权以后要在unix_detach_fds( )中通过unix_notinflight( )“销账”。所以，当全局量unix_tot_inflight非0时，就表示有这样的授权尚在“飞行”中，还没有被其目标进程所接收。同时，当通过sendmsg( )发送对已打开文件的授权时，在scm_send( )中会通过fget( )递增相应file结构中的共享计数，表示从这一刻起这个已打开文件已经多了一个“用户”。在目标进程接收到这个报文前，这个所谓用户就是内核。然后，当目标进程接收到这个报文时，要通过scm_recv( )调用scm_detach_fds( )。在里面先通过get_file( )

递增相应 file 结构中的共享计数，表示接收进程也成了这个已打开文件的用户，随后又通过 __scm_destroy( )递减这个 file 结构中的共享计数，表示内核不再是这个已打开文件的用户了。一旦载送着文件访问授权的报文被接收方进程所接收，共享这个已打开文件的用户就都是进程了，或迟或早这些进程总会关闭这个文件，而最后关闭这个文件的进程会将共享计数减至 0 而最终真正将其"关闭"。

可是，有两种情况要特别加以考虑。

第一种情况是我们这个插口的接收队列中有一个或几个报文载送着这样的文件访问授权。显然，由于正在关闭这个插口，这些报文都要被丢弃了。但是，所谓"丢弃"绝不是简单地释放缓冲区了事，因为对载送着文件访问授权的报文还得负起递减相应 file 结构中的共享计数的责任，否则这些文件永远也不可能真正关闭。我们回过去看一下前面的 402 行，接收队列中的报文（即 sk_buff 结构）是通过 kfree_skb( )释放的，我们已在前一节中看过它的代码。它最终会通过 sk_buff 结构中的函数指针 destructor 调用一个函数。这个函数一般都指向 sock_wfree( )，但是当报文载送着文件访问授权时则指向 unix_destruct_fds( )，这是在发送报文时由 unix_attach_fds( )设置的。函数 unix_destruct_fds( )的代码在 af_unix.c 中：

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( ) > unix_release( ) > unix_release_sock( ) > kfree_skb( ) > __kfree_skb( ) > unix_destruct_fds( )]

```
1123    static void unix_destruct_fds(struct sk_buff *skb)
1124    {
1125        struct scm_cookie scm;
1126        memset(&scm, 0, sizeof(scm));
1127        unix_detach_fds(&scm, skb);
1128
1129        /* Alas, it calls VFS */
1130        /* So fscking what? fput( ) had been SMP-safe since the last Summer */
1131        scm_destroy(&scm);
1132        sock_wfree(skb);
1133    }
```

先看这里的 unix_detach_fds( )，这也在 af_unix 中：

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( ) > unix_release( ) > unix_release_sock( ) > kfree_skb( ) > __kfree_skb( ) > unix_destruct_fds( ) > unix_detach_fds( )]

```
1111    static void unix_detach_fds(struct scm_cookie *scm, struct sk_buff *skb)
1112    {
1113        int i;
1114
1115        scm->fp = UNIXCB(skb).fp;
1116        skb->destructor = sock_wfree;
1117        UNIXCB(skb).fp = NULL;
1118
1119        for (i=scm->fp->count-1; i>=0; i--)
1120            unix_notinflight(scm->fp->fp[i]);
1121    }
```

这个函数的作用有三点，先是使 scm_cookie 结构中的指针 fp 指向随同 sk_buff 结构发送过来的 scm_fp_list 结构；其次将 sk_buff 结构中的函数指针 destructor 恢复成指向 sock_wfree；再就是为每个授权访问的文件通过 unix_notinflight( )“销账”。但是，至此还没有触及相应 file 结构中的共享计数，那是在 scm_destroy( )中完成的，有关的代码在 scm.h 和 scm.c 中：

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( ) > unix_release( ) > unix_release_sock( ) > kfree_skb( ) > __kfree_skb( ) > unix_destruct_fds( ) > scm_destroy( )]

```
27      static __inline__ void scm_destroy(struct scm_cookie *scm)
28      {
29          if (scm && scm->fp)
30              __scm_destroy(scm);
31      }
```

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( ) > unix_release( ) > unix_release_sock( ) > kfree_skb( ) > __kfree_skb( ) > unix_destruct_fds( ) > scm_destroy( ) > __scm_destroy( )]

```
101     void __scm_destroy(struct scm_cookie *scm)
102     {
103         struct scm_fp_list *fpl = scm->fp;
104         int i;
105
106         if (fpl) {
107             scm->fp = NULL;
108             for (i=fpl->count-1; i>=0; i--)
109                 fput(fpl->fp[i]);
110             kfree(fpl);
111         }
112     }
```

如前所述，这里的 fput( )递减相应 file 结构中的共享计数。当然，如果递减后成为 0 就将此文件最终关闭。

第二种情况就更加复杂了，那就是我们刚从将要关闭的插口发送了一个报文，将对此插口的访问授权给对方，但是该报文尚在“飞行”中，还没有被对方接收，而这一边的插口倒要关闭了。下面我们通过两个情景来说明可能存在的问题。

假定有两个进程 A 和 B，还有两个 Unix 域插口 sa 和 sb；并且 A 是 sa 惟一的用户，而 B 是 sb 惟一的用户。先来看第一个情景：

(1) A 通过 sa 发送一个报文到 sb，把对 sa 的访问授权给 B，因此已经递增了 sa 的 file 结构中的共享计数。该报文已经挂入到 sb 的接收队列中，所以 A 的 sendmsg( )操作已经成功返回。可是，B 没有在 sb 上接收报文，所以该报文还在 sb 的接收队列中等待被接收，对插口 sa 的访问授权还在“飞行”中。

(2) 接着，A 通过 close( )关闭 sa。内核递减 sa 的 file 结构中的共享计数，但是由于在递减前的计数为 2，所以递减后并未到达 0。因此，系统调用 close( )至此就完成了，并未对 sa 执行 sock_close( )。可是，进程 A 却从此失去了与 sa 的联系。

(3) B 没有从 sb 接收报文，但是却关闭了 sb。由于在此之前 sb 的 file 结构中的共享计数为 1，所以递减后达到了 0，因此内核对 sb 执行 sock_close( )。在执行 sock_close( )的过程中，内核对 sb 接收队列中的每个报文都执行 kfree_skb( )。当对上述由 A 发出的那个报文执行 kfree_skb( )时，在 scm_destroy( )中会对 sa 的 file 结构调用 fput( )。这时候，sa 的 file 结构的共享计数就变成了 0，从而引起 sock_close( )对 sa 的执行而将其最终关闭撤销了。最后，sb 插口本身也最终地关闭、撤销了。

当然，如果 B 接收了上述报文，那么 B 就成了 sa 的惟一用户。最后 B 终究要关闭 sa，那时 sa 也最终得到关闭和撤销。插口的 sa 和 sb 之间的这种关系可以推广到更多个插口而形成一条链，但是最终这条链中的所有插口都能正常地关闭和撤销。所以，这个情景没有什么问题。

再来看第二个情景：

(1) A 通过 sa 发送一个报文到 sb，把对 sa 的访问授权给 B。该报文已经挂入到 sb 的接收队列中，但尚未被接收，所以对 sa 的访问授权还在“飞行”中。

(2) B 也通过 sb 发送一个报文到 sa，把对 sb 的访问授权给 A。同样地，该报文已经挂入到 sa 的接收队列中，但尚未被接收，所以对 sb 的访问授权也还在“飞行”中。

(3) A 通过 close( )关闭 sa，但是因对 sa 的共享计数递减后未达到 0 而不能对 sa 执行 sock_close( )，可是进程 A 与插口 sa 之间的联系却已经切断了。

(4) B 也通过 close( )关闭 sb。同样地，因对 sb 的共享计数递减后未达到 0 而不能对 sb 执行 sock_close( )，可是进程 B 与插口 sb 之间的联系也已经切断了。

如果不采取措施的话，那么 A 发出的报文就会永远留在 sb 的接收队列中，而 B 发出的报文则会永远留在 sa 的接收队列中。两个插口的共享计数都是 1，却没有一个进程能访问到这两个插口的任何一个。插口本身是不能“生活自理”的，它自己不会关闭和撤销自己。于是，这两个插口，也就是有关的数据结构，连同接收队列中的所有报文都已成了“废料”，但却要永远占着位置不放。同样，插口 sa 和 sb 之间的这种关系也可以推广到更多个插口。如果说在前一种情景所涉及的插口构成一个链的话，那么在这个情景中所构成的是一个环，而区别也正在于此。这种现象与“死锁”的表现虽然不同，原理却是相通的。

对这一类问题的对策无非是“防”与“化”两个字。所谓“防”就是防止这种现象的发生；而“化”则首先要能检测到这种现象的发生，然后就来化解。具体到上述问题，“防”是很难的，所以只好在“化”字上做文章。为此目的，内核中设计了一个专门用来解决这个问题的“废料收集”机制，具体就是由前面 unix_release_sock( )代码中 426 行处调用的 unix_gc( )实现的。这个函数的代码在 net/unix/garbage.c 中，我们分段阅读：

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( ) > unix_release( ) > unix_release_sock( ) > unix_gc( )]

```
166	/* The external entry point: unix_gc( ) */
167
168	void unix_gc(void)
169	{
170		static DECLARE_MUTEX(unix_gc_sem);
171		int i;
172		unix_socket *s;
173		struct sk_buff_head hitlist;
```

```
174         struct sk_buff *skb;
175 
176         /*
177          *  Avoid a recursive GC.
178          */
179 
180         if (down_trylock(&unix_gc_sem))
181             return;
182 
183         read_lock(&unix_table_lock);
184 
185         forall_unix_sockets(i, s)
186         {
187             s->protinfo.af_unix.gc_tree=GC_ORPHAN;
188         }
189         /*
190          *  Everything is now marked
191          */
192 
193         /* Invariant to be maintained:
194             - everything unmarked is either:
195             -- (a) on the stack, or
196             -- (b) has all of its children unmarked
197             - everything on the stack is always unmarked
198             - nothing is ever pushed onto the stack twice, because:
199             -- nothing previously unmarked is ever pushed on the stack
200          */
201 
202         /*
203          *  Push root set
204          */
205 
206         forall_unix_sockets(i, s)
207         {
208             /*
209              *  If all instances of the descriptor are not
210              *  in flight we are in use.
211              */
212             if(s->socket && s->socket->file &&
213             file_count(s->socket->file)>atomic_read(&s->protinfo.af_unix.inflight))
214                 maybe_unmark_and_push(s);
215         }
216 
```

“废料”收集的过程是不容许两个进程同时进行的，所以要通过内核信号量 unix_gc_sem 把它保护起来。另一方面，在这个过程中要扫描和处理杂凑表 unix_socket_table[ ]中的所有队列，所以在此期间也不容许为任何其他目的来变动这些队列，因此还要将整个 unix_socket_table[ ]加上锁。代码中的

forall_unix_socket( )是个宏定义，其定义在 include/linux/af_unix.h 中：

```
16          #define forall_unix_sockets(i, s) \
17                  for (i=0; i<=UNIX_HASH_SIZE; i++) \
18                          for (s=unix_socket_table[i]; s; s=s->next)
```

显然，这是针对杂凑表中所有的队列，扫描队列中的所有 sock 结构。那么，扫描这些 sock 结构干什么呢？第一趟扫描时把所有的 sock 结构都标志成 GC_ORPHAN。也就是说，先假定所有的插口都已经成了前述“生活不能自理”的“孤儿”。Unix 域插口的 sock 结构中专为废料收集而设置了一个指针 profile.af_unix.gc_tree。只有 Unix 域插口才支持对已打开文件的授权，所以只有 Unix 域的插口才有废料收集的问题。常数 GC_ORPHAN 的定义也在 garbage.c 中：

```
85    /* Internal data structures and random procedures: */
86
87    #define GC_HEAD      ((unix_socket *)(-1))
88    #define GC_ORPHAN    ((unix_socket *)(-3))
89
90    static unix_socket *gc_current=GC_HEAD; /* stack of objects to mark */
91
92    atomic_t unix_tot_inflight = ATOMIC_INIT(0);
```

这里我们同时列出了其他几个定义。其中 unix_tot_inflight 就是当前正在“飞行”中的已打开文件访问授权的个数。而指针 gc_current 则在废料收集的过程中用来构筑一个“后进先出”队列。

顺便提一下，我们是在函数 unix_release_sock( )的末尾处调用 unix_gc( )的。此时我们要关闭并撤销的插口本身已经与杂凑表无关，它的 sock 结构在刚进入 unix_release_sock( )时就已经通过 unix_remove_socket( )从杂凑表的队列中摘除了。事实上，“废料收集”与正欲撤销的插口本身无关，只不过是乘机让它承担一点义务，作出一点贡献而已。

第一趟扫描将所有的插口都假定为“孤儿”，当然不见得就符合事实，所以第二趟扫描就要来加以甄别了。根据什么准则来甄别呢？那就是比较插口的 file 结构中的共享计数和 sock 结构中的另一个计数器 protoinfo.af_unix.inlight。我们以前看到过，当将对一个插口的访问授权通过 sendmsg( )发送给另一个进程时，要调用 unix_inflight( )递增该插口的 sock 结构中的这个计数器，以及全局性的计数器 unix_tot_inflight，而在对方接收了这个报文时则要调用 unix_notinflight( )递减这两个计数器。当然，对插口的 file 结构中的共享计数也要作类似的处理。如果我们比较这两个计数器的大小，则可以得到如下的结论：

(1) 文件共享计数 > 授权报文计数：正常。说明除正在“飞行”中的授权报文外至少还有一个用户，而这个用户必然是一个进程。所以，这个插口不是“孤儿”。

(2) 文件共享计数 = 授权报文计数：不正常。这个插口已经没有真正意义上的“用户”了，所以可能是“孤儿”。

(3) 文件共享计数 < 授权报文计数：不可能发生。

在第二趟扫描中，凡遇正常的 sock 结构就通过其指针 protinfo.af_unix.gc_tree 将其链入到队列 gc_current 的前部，或者说将其推入“堆栈”gc_current。实现这一操作的函数代码如下（见 garbage.c）：

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( ) > unix_release( ) > unix_release_sock( )

> unix_gc( ) > maybe_unmark_and_push( )]

```
156    extern inline void maybe_unmark_and_push(unix_socket *x)
157    {
158        if (x->protinfo.af_unix.gc_tree != GC_ORPHAN)
159            return;
160        sock_hold(x);
161        x->protinfo.af_unix.gc_tree = gc_current;
162        gc_current = x;
163    }
```

注意，将一个 sock 结构链入到 gc_current 队列中，并不意味着这个结构就脱离了杂凑表 unix_socket_table[ ]中的队列，sock 结构可以通过不同的指针链入不同的队列。

这样，第二趟扫描以后，凡是其指针 protinfo.af_unix.gc_tree 仍为 GC_ORPHAN 的插口很可能就是"孤儿"了。但是，至此还不能肯定所有这些插口都已经成了"孤儿"，因为这实际上取决于授权的对象，更确切地说是接收授权报文的插口。以上面所举例子中的插口 sa 为例。对 sa 的访问授权有可能已经通过多个报文发送到了多个插口，然后 A 关闭了 sa。只要有一个对访问 sa 的授权报文是在一个"正常"插口的接收队列中，那么这个报文就有希望被某个进程接收，从而使 sa 又有了真正意义上的用户，那个进程自会负起最后关闭并撤销 sa 的责任。所以，还要进一步甄别，这一次是检查所有"正常"插口的接收队列中的每一个报文，而所有"正常"插口的 sock 结构都已经在队列 gc_current 中了。

继续看 unix_gc 的代码：

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( ) > unix_release( ) > nix_release_sock( ) > unix_gc( )]

```
217        /*
218         *  Mark phase
219         */
220
221        while (!empty_stack( ))
222        {
223            unix_socket *x = pop_stack( );
224            unix_socket *sk;
225
226            spin_lock(&x->receive_queue.lock);
227            skb=skb_peek(&x->receive_queue);
228
229            /*
230             *  Loop through all but first born
231             */
232
233            while(skb && skb != (struct sk_buff *)&x->receive_queue)
234            {
235                /*
236                 *  Do we have file descriptors ?
```

```
237                 */
238                if(UNIXCB(skb).fp)
239                {
240                    /*
241                     *  Process the descriptors of this socket
242                     */
243                    int nfd=UNIXCB(skb).fp->count;
244                    struct file **fp = UNIXCB(skb).fp->fp;
245                    while(nfd--)
246                    {
247                        /*
248                         *  Get the socket the fd matches if
249                         *  it indeed does so
250                         */
251                        if((sk=unix_get_socket(*fp++))!=NULL)
252                        {
253                            maybe_unmark_and_push(sk);
254                        }
255                    }
256                }
257                /* We have to scan not-yet-accepted ones too */
258                if (x->state == TCP_LISTEN) {
259                    maybe_unmark_and_push(skb->sk);
260                }
261                skb=skb->next;
262            }
263            spin_unlock(&x->receive_queue.lock);
264            sock_put(x);
265        }
266
```

我们把这一段代码留给读者自己阅读。这里的 pop_stack( )从队列 gc_current 的前部摘下一个 sock 结构（所以是“后进先出”），而 empty_stack( )则测试该队列的长度，函数 skb_peek( )从一个接收队列中取第一个报文（即 sk_buff 结构）的指针，但并不将其摘下。

此处给出 pop_stack( )等的代码如下（garbage.c）：

```
140    /*
141     *  Garbage Collector Support Functions
142     */
143
144    extern inline unix_socket *pop_stack(void)
145    {
146        unix_socket *p=gc_current;
147        gc_current = p->protinfo.af_unix.gc_tree;
148        return p;
149    }
150
```

```
151	extern inline int empty_stack(void)
152	{
153		return gc_current == GC_HEAD;
154	}
```

注意，maybe_unmark_and_push( )对于已经进入 gc_current 队列的 sock 结构不起作用（见 159 行）。

经过这两遍甄别以后，所有仍是 GC_ORPHAN 的插口就最后定性为“孤儿”了。对这些“孤儿”怎么办呢？扫描其接收队列中的所有报文，凡发现载有访问授权的报文就将其从接收队列中摘下来集中到一个临时的队列 hitlist 中，最后对这些报文执行 kfree_skb( )，其代码我们已在前面看到过。注意这里只要对载有访问授权的报文调用 kfree_skb( )就行了，对于已定性为“孤儿”的 sock 结构并不需要特地加以处理，因为这一定已经隐含在对某个报文的 kfree_skb( )中了。至于“孤儿”sock 结构的接收队列中的其他报文，也自会得到释放（请读者想一下，为什么。）

最后给出 unix_gc 代码的余下部分：

[sys_close( ) > filp_close( ) > fput( ) > sock_close( ) > sock_release( ) > unix_release( ) > nix_release_sock( ) > unix_gc( )]

```
267		skb_queue_head_init(&hitlist);
268	
269		forall_unix_sockets(i, s)
270		{
271			if (s->protinfo.af_unix.gc_tree == GC_ORPHAN)
272			{
273				struct sk_buff *nextsk;
274				spin_lock(&s->receive_queue.lock);
275				skb=skb_peek(&s->receive_queue);
276				while(skb && skb != (struct sk_buff *)&s->receive_queue)
277				{
278					nextsk=skb->next;
279					/*
280					 *  Do we have file descriptors ?
281					 */
282					if(UNIXCB(skb).fp)
283					{
284						__skb_unlink(skb, skb->list);
285						skb_queue_tail(&hitlist,skb);
286					}
287					skb=nextsk;
288				}
289				spin_unlock(&s->receive_queue.lock);
290			}
291			s->protinfo.af_unix.gc_tree = GC_ORPHAN;
292		}
293		read_unlock(&unix_table_lock);
294	
295		/*
```

```
296          * Here we are. Hitlist is filled. Die.
297          */
298
299         while ((skb=__skb_dequeue(&hitlist))!=NULL) {
300             kfree_skb(skb);
301         }
302
303         up(&unix_gc_sem);
304     }
```

如前所述，对 unix_gc( )的调用其实并不是非得放在 unix_release_sock( )中。例如，完全可以周期性地调用这个函数。但是，放在 unix_release_sock( )的末尾还是比较合适的。也许读者会问，为什么不把对这个问题的处理放在系统调用 close( )的开头呢？例如，在递降了 file 结构中的共享计数以后，如果没有达到 0 就与 sock 结构中的授权计数比较一下，这样马上就可以知道是否有本插口发生的授权报文正在“飞行”中，这样不是更有效吗？问题在于，文件系统操作界面的函数都是通用的，必须适用于所有不同类型的文件，而插口只是其中的一种。以关闭文件为例，内核中由 sys_close( )调用 filp_close( )，再由 filp_close( )调用 fput( )，所有这些函数都只是在 vfs 层对抽象的文件进行操作，根本就不知道所操作的到底是个什么样的文件，所以显然是不合适的。

7.9 其他

如前所述，“插口“机制有两个用户程序界面，一个是为插口专设的，另一个就是通用的文件操作界面。这样，即使只考虑 Unix 域的插口，内核中与之有关的函数就自然更多一些了。我们已经把主要的和比较复杂的 Unix 域的插口操作及其代码作了详细介绍，但是限于篇幅我们只好把剩下的一些代码留给读者自己阅读了。

从插口操作界面，也就是系统调用 socketcall( )在内核中的入口 sys_socketcall( )的角度来看，留下来的有这么一些：sys_getsockname( )、sys_getpeername( )、sys_socketpair( )、sys_shutdown( )、sys_getsockopt( )以及 sys_setsockopt( )。这些函数也要通过数据结构 unix_dgram_ops 或 unix_stream_ops 跳转到 unix 域插口对这些操作的实现，分别为 unix_getname( )、unix_socketpair( )、unix_shutdown( )。函数 sys_getpeername( )与 sys_getsockname( )的不同之处在于后者是取插口本身的地址，而前者是取对方插口的地址（限于已调用 connect( )以后）。所以最终都是由 sys_getname( )完成的。另外，sys_getsockopt( )和 sys_setsockopt( )二者都是为网络环境设计的，因为在具体的网络通信规程中常常有许多可选项要设置，而在 unix 域中则不存在这个问题。所以，在 unix_dgram_ops 和 unix_stream_ops 两个结构中，有关的函数指针为 sock_no_getsockopt( )和 sock_no_setsockopt( )，这些函数都只是返回一个出错代码－EOPNOTSUPP，表示系统不支持这些操作。

总的来说，剩下的这些函数都是很简单、很直截了当的。例如，unix_socketpair( )的代码为（见 af_unix.c）：

[sys_socketcall( ) > sys_socketpair( ) > unix_socketpair( )]

```
1015    static int unix_socketpair(struct socket *socka, struct socket *sockb)
```

```
1016    {
1017        struct sock *ska=socka->sk, *skb = sockb->sk;
1018
1019        /* Join our sockets back to back */
1020        sock_hold(ska);
1021        sock_hold(skb);
1022        unix_peer(ska)=skb;
1023        unix_peer(skb)=ska;
1024        ska->peercred.pid = skb->peercred.pid = current->pid;
1025        ska->peercred.uid = skb->peercred.uid = current->euid;
1026        ska->peercred.gid = skb->peercred.gid = current->egid;
1027
1028        if (ska->type != SOCK_DGRAM)
1029        {
1030            ska->state=TCP_ESTABLISHED;
1031            skb->state=TCP_ESTABLISHED;
1032            socka->state=SS_CONNECTED;
1033            sockb->state=SS_CONNECTED;
1034        }
1035        return 0;
1036    }
```

参数 socka 和 sockb 为已经在 sys_socketpair( )中通过 sock_create( )创建好的两个 socket 数据结构。这里的 1022 行和 1023 行使它们的 sock 结构各自指向对方，如果是“有连接”插口则还要设置好各自的有限状态以及插口的状态。

另一方面，从文件系统的操作界面来看，则还有这么几个函数：sock_lseek( )、sock_poll( )、sock_ioctl( )、sock_mmap( )以及 sock_fasync( )。其中 sock_lseek( )只是返回一个出错代码，因为插口并不支持 lseek( )。函数 sock_poll( )用于系统调用 select( )；sock_ioctl( )用于系统调用 ioctl( )。相应的 unix 域函数为 unix_poll( )、datagram_poll( )以及 unix_ioctl( )。读者可以结合 “文件系统”和“字符设备驱动”这两章把它们读懂。操作 sock_mmap( )在 Unix 域中的实现为 sock_no_mmap( )，就是说 Unix 域插口并不支持 mmap( )。最后，sock_fasync( )用来让插口在每当有报文（包括连接请求）到达时就向若干进程发出 IOSIG 信号，从而实现对报文的异步接收。这个函数并没有特别针对 Unix 域插口的底层操作，而是通用于所有插口。我们在讲述 sock_close( ) 时已经列出了 sock_fasync( )的代码，但是只讲了当调用参数 on 为 0 时，也就是要停止异步接收时的那部分代码。不过，读懂了一个方向的操作（停止异步接收）的代码以后，要读懂其相反方向（启动异步接收）的代码应该是比较容易的了。

最后，我们建议读者把 sock 数据结构（以及 sk_buff）的定义打印出来，再根据我们讲到的内容，试着为数据结构中的每个字段（成分）（只要是已经讲到过的）的用途和作用加上注释。如果你觉得很多字段的用途和作用实在不是三言两语能够讲清的，你就明白了为什么我们没有在一开始就列举该数据结构各个字段的作用了。在这种情况下，你不妨在打印的清单上写上“见××页”等字样，以备日后查阅。

第8章

设备驱动

8.1 概述

设备驱动在系统中的重要地位是无需多加说明的，计算机最基本的三个物质基础就是CPU、内存以及输入/输出（I/O）设备。严格地说，离开了对设备的操作，即输入/输出操作，计算机本身也就失去了意义。相比之下，文件系统的存在只不过是使对设备的操作更为方便、更为有效、更有组织、更接近人类的思维方式而已。所以，文件操作是对设备操作的组织与抽象，而设备操作则是对文件操作的最终实现。

Unix操作系统从一开始就将所有的设备（而不仅是磁盘上的文件）全都看成文件，都纳入文件系统的范畴，都通过文件操作的界面进行操作。这就意味着：

- 每一项设备都至少由文件系统中的一个文件（更确切地说是节点）代表，因而都有一个“文件名”。每个这样的“设备文件”都惟一地确定了系统中的一项设备。应用程序通过设备的文件名寻访具体的设备，而设备则像普通文件一样受到文件系统访问权限控制机制的保护。
- 应用程序通常可以通过系统调用open( )“打开”设备文件，建立起与目标设备的连接，或曰“上下文”。代表着该设备的文件节点中记载着建立这种连接所需的信息。对于执行该应用程序的进程而言，建立起的连接就表现为一个已打开文件。
- 打开了代表着目标设备的文件，即建立起与设备的连接以后，就可以通过read( )、write( )、ioctl( )等常规的文件操作对目标设备进行操作。从应用程序的角度看，设备文件逻辑上的空间是个线性空间。从这个逻辑空间到具体设备的物理空间的映射则由内核提供，并划分成文件操作与设备驱动两个层次。

这样，对于一个具体的设备来说，文件操作和设备驱动就成为同一事物的不同层次，而不是互相独立或平行的两个概念。从这种观点和结构模型出发，一般而言，至少可以在概念上把一个系统划分成应用、文件系统以及设备驱动这么三个层次。这不仅适用于Unix，也可以运用到其他的操作系统，尽管具体的划分和安排可以不同。例如，通常文件系统层和设备驱动层都在内核中，但是也有的系统把文件系统放在内核外面作为应用层的一个进程。表面上看来（从进程的角度）此时文件系统与应用

程序互相平行，但是如果考察对一具体文件的操作过程，就可以看出用来实现文件系统的进程实际上成了应用程序与内核之间的一个附加的层次。有些系统只提供设备驱动而不提供文件系统，由应用程序在进程内部实现其自己的文件系统(例如通过一些库函数调用)。显然这只是文件系统层的物理位置不同，而并不改变整个结构模型的层次结构。即使有些特殊的应用程序根本不使用文件系统，而直接对设备驱动层操作，那也可以认为只是一种特例，即此时的文件系统是完全“透明”的。还有些操作系统，特别是早期的操作系统，把设备驱动做成与文件系统平行，有独立的命名空间以及一套独立的操作(系统调用)，让应用层直接与设备驱动打交道。表面上看来这是完全不同的结构模型，但是如果作更深入的思考就可以领悟到：实际上这只不过是实现方法上的不同，只不过是为设备驱动保留了更多的特殊性，对设备少作了某些抽象而已，而采用一套独立的、专用的操作则有损其灵活性。Unix 的设计者正是认识到了这一点，而在 Unix 中首先明确地把设备驱动纳入了文件系统。所以，许多系统的外观也许不同，但是如果加以仔细考察就可以发现实际上说到底还是同一个结构模型。而 Unix(和 Linux)的文件系统，则是这种结构模型的典型实现，也许可以说是最自然的实现。本书上册第 5 章“文件系统”的文件系统结构图（图 5.1）就反映了这种系统结构，其文件系统层和设备驱动层都在内核中。从这个意义上说，本书第 5 章也许不应该叫“文件系统”，而应更确切地称为“磁盘文件系统”或“普通文件系统”。所以说，“文件系统”这个词的含义是很模糊的。

但是，话虽如此，对于不同的设备其文件系统层的“厚度”却有所不同。对于像磁盘这样结构性很强，并且其内容需要进一步加以组织和抽象的设备来说，其文件系统很“厚”很“重”，这一点读者在阅读“文件系统”一章时想必已有所体会。磁盘设备的复杂性来自两个方面。一方面是介质本身的结构，如“磁道”、“柱面”、“扇区”以及抽象意义上的“记录块”；另一方面是在“记录块”基础上的又一层组织和抽象，即“磁盘文件”。这样，在物理介质上的第一层抽象使操作者不必关心读/写的物理位置究竟在哪一个磁道，哪一个扇区；而第二层抽象则使操作者不必关心读/写的内容在哪一个逻辑“记录块”中。很自然地，我们把第一层抽象归入设备驱动层，而第二层抽象则归入文件系统层。但是，还有一些设备，如字符终端、字符打印机等，则由于本身并没有什么结构，甚至不存在存储介质，因而简单得多。对于这样的设备，其文件系统层自然就比较“薄”，甚至近乎“透明”。不光是文件系统层比较薄，连设备驱动层也可能很简单。另一方面，即使是对于磁盘设备，在有些应用中也可以绕过第二层抽象而简单地把它当成记录块“数组”，也就是忽略由这些记录块的内容形成的关连和组织，使原来很厚的文件系统层变得很薄。在这种情况下，我们称之为“原始”（raw）设备。当然，物理上还是同一个设备，只不过是“横看成岭侧成峰”——从不同的角度看问题而已。在这方面，将设备驱动纳入文件系统又表现出其优越性，因为对同一设备的不同驱动方式可以由不同的文件来代表。下页的示意图（图 8.1）或许将有助于读者的理解。

在这个示意图中，处于应用层中的进程通过“打开文件号”fd 与已打开文件的 file 结构相联系，每个 file 结构代表着对一个已打开文件操作的上下文。通过各个上下文，进程使用各个文件的线性逻辑空间进行文件操作。对于普通文件，即“磁盘文件”，文件的逻辑空间在文件系统层内按具体文件系统的结构和规则映射到设备的线性逻辑空间，然后在设备驱动层进一步从设备的逻辑空间映射到其物理空间。这样，一共经历了两层映射。或者，也可以反过来说，磁盘设备的物理空间经过两层抽象而成为普通文件的线性逻辑空间。而对于“设备文件”，则文件的逻辑空间通常直接就等价于设备的逻辑空间，所以在文件系统层就不需要有映射。但是，也有些设备需要在文件系统层中有一些简单的映射。从图中还可以看出，对同一个设备也可以通过不同的文件以不同的方式来操作。这里还要指出，代表

着设备文件的节点都要通过某个索引节点才能寻访，而对索引节点又要通过一些目录节点才能寻访，这些目录节点实质上相当于普通文件，所以在打开设备文件的过程中即隐含着对普通文件的操作。

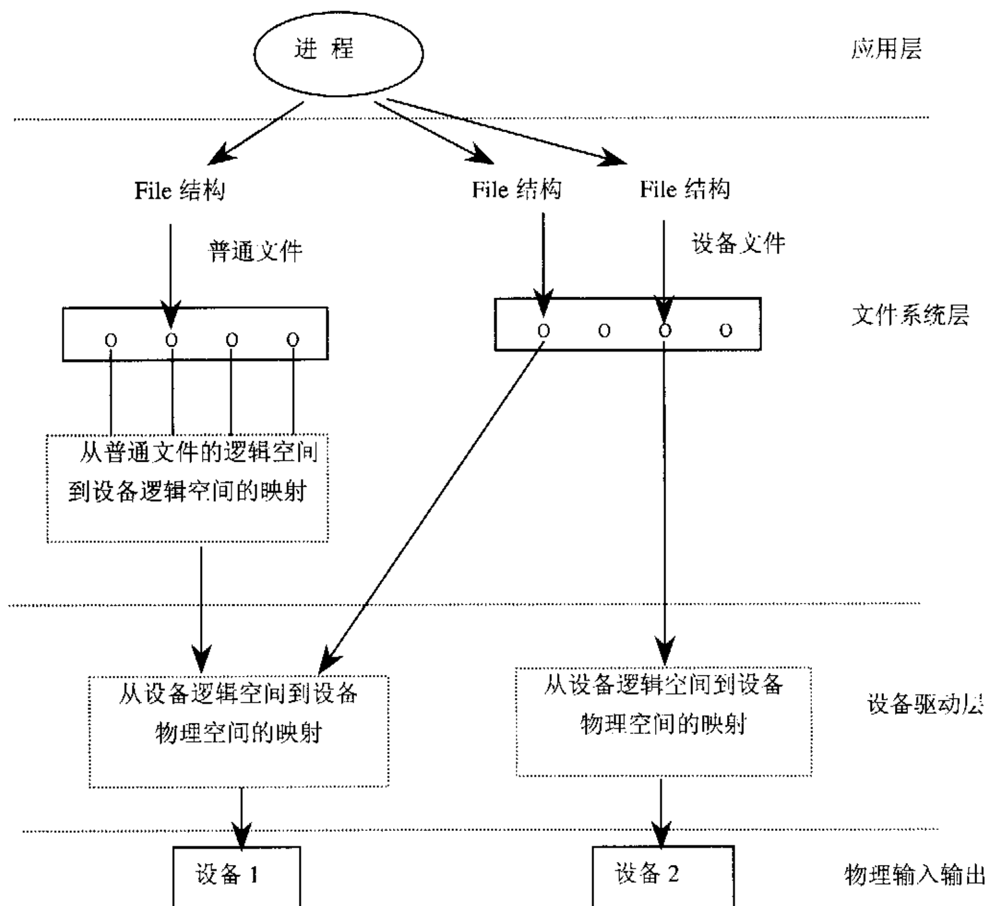


图 8.1　设备驱动分层结构示意图

Unix(以及 Linux)将设备分成两大类。一类是像磁盘那样以记录块或“扇区”为单位，成块进行输入/输出的设备，称为“块设备”；另一类是像键盘那样以字符（字节）为单位，逐个进行输入/输出的设备，称为“字符设备”。文件系统通常都建立在块设备上，但是也并无规定说在字符设备上就不可以建立文件系统。通过数十年的发展，块设备和字符设备之间的界线已经模糊了，但还是沿用着这样的划分。两类设备之间的界线之所以变得模糊，一方面是由于传统字符设备变得愈来愈复杂了，另一方面是出现了一些既像是块设备又像是字符设备的新设备。例如，网络接口卡就是这样的设备，说它是字符设备吧，它的输入/输出却是有结构的、成块的（报文、包、帧）；说它是块设备吧，它的“块”又不是固定大小的，大到数百甚至数千字节，小到几个字节。块设备与字符设备还有个区别，那就是块设备的介质必须是存储介质，并且支持“随机访问”（对指定地址的访问），因而系统调用 lseek( )对于块设备是不可缺少的操作；而字符设备的介质则一般都是传输介质，一般只支持“顺序访问”，因而 lseek( )对于字符设备意义不大。从这个意义上讲，网络设备当然就是字符设备了。不过这也不是绝对

的。举例来说，在 Unix 系统中块设备都有对应的“原始设备”。例如，有块设备/dev/hda，则必有另一设备/dev/rhda，这里的字符“r”表示“raw”即“原始”的意思。原始设备都是作为字符设备对待的，虽然它们的介质仍为存储介质，并且也支持随机访问。（在 Linux 中，不再为块设备提供原始设备文件，因为块设备在安装之前实际上就相当于原始状态）。所以，一般都只是把成块输入/输出并且打算在上面建立普通文件系统的设备才称为“块设备”。

对设备的这种类别划分并不是一个理论问题，而是个技术问题。为什么呢？前面讲过，在代表着设备的文件节点中记载着与特定设备建立连接所需的信息。这种信息由三部分构成：第一部分是文件（包括设备）的类型，第二部分是一个“主设备号”，第三部分是一个“次设备号”。其中设备类型和主设备号结合在一起惟一地确定了设备的驱动程序及其界面，而次设备号则说明目标设备是同类设备中的第几个。例如，当主设备号为 2 时，若设备类型为块设备就是指软盘驱动器，而若为字符设备则是指所谓“伪终端”（Pseudo TTY）设备。这样的安排从 Unix 的早期一直延用至今，出于兼容的考虑又不宜轻易变动，所以每项设备都得非此即彼，或划入块设备，或划入字符设备，尽管有些设备确实非驴非马。

从早期 Unix 以来一直沿用 8 位的主设备号，这就把块设备和字符设备的种类都限制到了 256 种，现在已经感到不够用了。就字符设备而言，在 Linux 源代码的 Documentation/devices.txt 中已经将主设备号分配到了 194 号。另一方面，从早期 Unix 开始就把所有的设备文件（节点）都放在/dev 目录中，使这个目录成为一个“平面”的，而不是树状的有层次的目录。当目录中的节点数量很多时，就会使打开这些文件时的效率受到影响。而且，在风格上也与文件系统中其他的部分不一致。所以，人们提出了改进的方案，就是把设备文件都放在一个树状的特殊文件系统 devfs 中。显然，这种新的方案比原来的要好，而且在现有文件系统的框架中也很容易实现。但是，新的方案必须与原有的兼容，使已有的应用软件可以继续运行。与 devfs 有关的代码都在 fs/devfs 中，我们也专门写了一节加以介绍。不过，我们有时候还会以原来平面结构的/dev 作为实例。这是因为目前老的方案还占着主导地位，机理也比较简单，再说新的 devfs 也与之兼容。读者可以在理解了基于主/次设备号的设备驱动以后再结合 devfs 一节加深理解。

要使一项设备在系统中成为可见，成为应用程序可以访问的设备，首先当然是要在文件系统中有一个代表此项设备的文件节点，这是通过系统调用 mknod( )实现的。可是，更重要的是在设备驱动层中要有这种设备的驱动程序。在早期的操作系统中（不仅是 Unix），设备驱动程序都静态地连接在内核中。可是，把所有可能的设备驱动程序都连接到内核中显然是不现实的，因为那样会使内核的体积大到不合理、甚至根本无法运行的程度。所以，通常都为最终用户提供一个“系统生成”的工具和手段，让用户根据具体的需要挑选一些已经事先编译好的模块，同时由系统生成工具对以高级语言编写的“设备表”加以修改，在表中加入相应驱动程序的函数指针，然后加以编译并将所挑选的模块与内核连接在一起。

这种方式给用户带来了很多不便，因为每当要往系统中增加或更换一种设备时就必须再进行一次系统生成。同时，随着网络技术的发展，不断需要将新出现的网络规程以设备驱动的形式在内核中实现，而静态连接的设备驱动程序同样为内核在这方面的功能更新和升级带来不便。Unix 在这方面的不足在一个时期内阻碍了它的进一步普及。相比之下，曾经在世界范围内得到广泛采用的 DOS 操作系统虽然简陋，但是却提供了一种手段称为 TSR（Terminate and Stay Resident），让用户程序可以动态地(在系统运行时)把一段可执行程序“粘贴”到操作系统上，并提供了系统调用让用户程序可以改变中断向

量。当然，TSR 给系统的安全性带来了极大的问题，而且对于像 Unix 这种多用户、多进程的系统它根本就不现实。但是不可否认这对于 DOS 以及 Windows（直到 Windows 3.1）的普及，乃至 PC 的普及起了相当大的作用。因为正是由于这样，许多 PC 兼容机和外部设备的厂家才能随时向用户提供更加新型的硬件，同时提供一张软盘让用户动态地“安装”新的驱动程序。

Unix 很早就支持所谓“共享库程序”，或称动态连接程序库，这在一定程度上缓解了这个问题，可是并不能从根本上解决问题，因为动态连接的库程序是在用户空间运行的（DOS 根本就不分用户空间和系统空间）。确实，对设备的许多操作可以在用户空间中实现，而只把物理的输入和输出留给内核。但是那样一来，设备驱动程序和设备之间的结合就不紧密，从而使效率降低甚至根本不能运转。另一方面，对于像 Unix(以及 Linux)这样的系统，当然不能照搬 TSR 一类的机制（Windows 95/98，Windows NT 都已不再使用 TSR）。设计出一种既能在多用户、多进程环境下动态地安装设备驱动程序，又能保持内核安全的机制，就成为对设计人员的一种挑战。

可安装模块正是在这样一种背景下产生的。可安装模块（module）是经过编译但尚未连接的目标代码（.o）文件，可以在系统运行时动态地“安装”到内核中。这种动态安装既可以由特权用户进行，也可以由内核在有需要时自动地启动（设想在网络环境下接收到了一个特殊的报文，而实现这种报文规程的模块尚未装入）。在内核的源代码中可以规定允许“移出”（export）内核的一些符号，如子程序入口、全局变量等，使可安装模块可以在程序中调用这些内核子程序或访问这些全局变量。对这些符号的引用是在模块安装的时候连接解决的，所以安装的过程实际上就包括了连接的过程。由于连接的部分对象已经装入内存并且已在运行，所以是动态连接。例如，我们在第 3 章中讲到内核中有个全局量 jiffies，这个变量在每次时钟中断时都要递增，因此可以用作基本的计时手段。假定在一个可安装模块中需要访问这个变量，就可以在其源代码中把它说明为“外部”（extern）变量。在编译时，gcc 知道这是一个需要在连接时解决的外部符号，但是并不知道这个变量在哪里，所以就将其地址暂时空着，留待连接时再来填写。到安装模块时，应用程序可以通过系统调用向内核查询变量 jiffies 所在的位置，只要这个变量是允许移出的，内核就会将其地址返回给应用程序。而应用程序则将其地址填入该模块中所有访问这个变量的指令内以及其他所有要用到其地址的地方。这样，变量 jiffies 就“连接”上了。

这些符号连接操作是由一个实用程序/sbin/insmod 完成的。这个实用程序不但负责模块与内核的连接，也负责把模块的目标文件（.o 文件）“装入”到内核空间。所以，insmod 的功能与 ld 相似，但是 insmod 所进行的是动态的(运行时的)连接和装入，而 ld 所进行的则是静态的连接和装入。与/sbin/insmod 相对应，还有一个实用程序/sbin/rmmod，其作用是将一个已安装的模块从内核中拆除。当然，像 insmod 一样，只有特权用户才能执行 rmmod。

除这两个实用程序外，Linux 为此而增设了若干系统调用，可安装模块机制主要就是通过这些系统调用实现的。在本章中，我们将与读者一起阅读这些系统调用的代码。

在这里，我们不妨将可安装模块与 TSR 程序作一简单的比较（尽管这二者实际上并没有多大的可比性）。在 TSR 程序中，是无法与内核中的符号（子程序、全局量）作符号连接的，而只能根据一些约定（将这些变量放在指定的地址上）来访问一些重要的全局量，以及通过陷阱指令调用由 BIOS 提供的功能和操作。例如 INT 9 为显示器屏幕操作，INT 13 为磁盘操作，还有就是通过 INT21 进行的 DOS 系统调用，等等。这样，一方面内核中的很多资源（除系统调用和一些约定的变量地址外）并未向 TSR 程序开放，而另一方面则又完全不设防，一个 TSR 程序完全可以为所欲为。例如，在某个条件满足时突然通过 INT13 将整个磁盘上的数据全部写成 0。光凭这一点，一些病毒的肆虐和恶劣表现就不足为

奇了。而在 Linux 的可安装模块中，由于 Linux 在运行时与 BIOS 没有关系（仅在引导时用到 BIOS），一方面凡是由内核“移出”的所有符号都可以在模块中引用，从而扩大了内核向可安装模块开放的资源范围；另一方面，除这些特意移出的符号以及系统调用以外，应用程序就别无途径可以直接访问内核中的资源。而系统调用，特别是用于文件系统操作的系统调用，是受到访问权限机制的严格控制的。这样，向可安装模块开放的子程序和全局量都经过仔细的权衡，即使不能保证这些资源全都是“无毒”的，也还有最后一道防线，因为只有特权用户才能安装这些模块。而且，由于 Linux 内核的源代码是公开的，有特殊要求的用户还可以对移出符号的名单加以调整。例如，要是知道不需要安装有关磁盘的模块，就可以将 ll_rw_block( )等有关块设备驱动的函数从内核的“移出名单”中剔除。目前，可安装模块机制是作为内核的一个可选项提供的，用户可以在编译内核时通过 CONFG_MODULES 选择包含或排除对这种机制的支持。

设备驱动层是直接与物理设备打交道的，在实际的实现中则因系统的结构和具体设备的物理特性而有不同的驱动方式。事实上，多数设备都是“中断驱动”的，而块设备往往都采用 DMA（“直接访问内存”）方式，所以物理设备的输入/输出从本质上说大多是异步的。相比之下，文件操作既可以是同步的，也可以是异步的，但是多数情况下是同步的。以从键盘读一个字符的过程为例，用户进程通过系统调用 read( )企图从标准输入文件读一个字符，但是真正的、物理意义上的从键盘读入通常并不是发生在用户进程调用 read( )的瞬间。如果在此之前用户已经按了键，那么所按的字符已经通过中断服务程序读了进来，放在缓冲区中等待由进程读取，此时上述 read( )操作立刻就可以完成而返回这个字符。但是，如果缓冲区中没有字符可读，那当前进程通常就要睡眠等待。等待到什么时候呢？等待到用户按键的时候，那是异步的，也就是无法预测何时会发生的。从这个意义上说，设备驱动程序是上层的同步操作与底层的异步操作之间的桥梁。

设备驱动程序要直接访问外部设备或其接口卡上的物理电路，这部分电路通常都是以“寄存器”形式出现的。根据访问外部设备寄存器的不同方式，可以把 CPU 分成两大类。一类 CPU 把这些寄存器看作内存的一部分，寄存器参与内存统一编址，访问寄存器就通过一般的访问内存指令进行，所以这种 CPU 没有专门用于设备 I/O 的指令。这样的 CPU 有 M68K、Power PC 等等。另一类 CPU 将外部设备的寄存器看成一个独立的地址空间，所以访问内存指令不能用来访问这些寄存器，而要为对外部设备寄存器的读/写单独设置专用的指令，如 in、out 等。i386 就是属于这一类的 CPU。可想而知，跟这些寄存器直接有关的代码是因处理器而异的。

就 i386 结构的 CPU 而言，用于外部设备寄存器读/写的指令主要就是两条，即 in 和 out。但是，像访问内存指令一样，根据读/写的对象为字节、字或长字而有 inb/inw/inl 和 outb/outw/outl 等变形。不过，Linux 内核的代码中一般都不直接使用这些指令，而将这些函数“包装”在一些相应的函数中，这些函数有 inb( )、outb( )、inw( )、outw( )等。

读者在第 3 章中曾经看到，内核中有些函数是利用 C 语言的编译时字符串拼接功能、由 gcc 在预处理阶段生成出来的，所以如果通过字符串搜索工具(如 grep)在源代码中搜索这些函数的定义就会找不到。这里，inb( )、outb( )这些函数的定义也是这样，是由 gcc 在编译时的预处理阶段根据一些宏定义生成出来的。下面我们就来看这些函数的生成，有关的代码都在 include/asm-i386/io.h 中。

先看 outb( )/outw( )/outl( )的生成：

```
92    __OUT(b,"b",char)
93    __OUT(w,"w",short)
```

```
94      __OUT(l,,int)
```

这里的__OUT( )是宏定义，其代码为：

```
58      #define __OUT(s,s1,x) \
59      __OUT1(s,x)     OUT2(s,s1,"w") : : "a" (value), "Nd" (port)); } \
60      __OUT1(s##_p,x)     OUT2(s,s1,"w")   \
                  __FULL SLOW_DOWN_IO : : "a" (value), "Nd" (port));} \
```

再看宏定义__OUT1、__OUT2 以及__FULL_SLOW_DOWN_IO：

```
52      #define __OUT1(s,x) \
53      extern inline void out##s(unsigned x value, unsigned short port) {
54
55      #define __OUT2(s,s1,s2) \
56      __asm__   volatile__ ("out" #s " %" s1 "0,%" s2 "1"

46      #define __FULL_SLOW_DOWN_IO __SLOW_DOWN_IO

38      #define __SLOW_DOWN_IO "\njmp 1f\n1:\tjmp 1f\n1:"
```

经过 gcc 的预处理以后，以上面的 92 行为例，就变成了这样：

```
extern inline void outb(unsigned char value, unsigned short port) {
    __asm     volatile__ ("outb  %b0, %w1" : : "a" (value), "Nd" (port)); }

extern inline void outb_p(unsigned char value, unsigned short port) {
    __asm__ __volatile   ("outb  %b0, %w1"
        "jmp 1f"
        "1: jmp 1f"
        "1:"
        : : "a" (value), "Nd" (port)); }
```

这里生成了两个函数，一个是 outb( )，还有一个是 outb_p( )。区别在于 outb_p( )中在输出指令以后有意通过两条 jmp 指令引入了一些延迟。有些外设寄存器的速度比较慢，在写入以后需要一些恢复时间。如果 CPU 的速度太快，就有可能在对同一寄存器的连续两次写操作之间隔得太紧而使寄存器来不及恢复，在这种情况下就应该调用 outb_p( )而不是 outb( )，使寄存器有时间恢复。此外，汇编指令中的%b0 和%w1 表示%0 的宽度为 8 位，而%1 的宽度为 16 位。还可以看出，%b0 中的 b 是从 92 行传下来的，而%w1 中的 w 则是固定的(见 59 和 60 行)，因为外设寄存器的地址是 16 位的。此外，变量 value 与寄存器%eax 结合，port 与%edx 结合。

同样，根据 93 和 94 行会分别生成出 outw( )、outw_p( )、outl( )以及 outl_p( )。

再看用于从寄存器输入的 inb( )、inw( )、inl( )等函数的生成。

```
82      #define RETURN_TYPE unsigned char
83      __IN(b,"")
```

```
84     #undef RETURN_TYPE
85     #define RETURN_TYPE unsigned short
86     __IN(w,"")
87     #undef RETURN TYPE
88     #define RETURN_TYPE unsigned int
89     __IN(1,"")
90     #undef RETURN TYPE

62     #define __IN1(s) \
63     extern inline RETURN_TYPE in##s(unsigned short port) { RETURN_TYPE _v;
64
65     #define __IN2(s,s1,s2) \
66     __asm__ __volatile__ ("in" #s " %" s2 "1,%" s1 "0"
67
68     #define __IN(s,s1,i...) \
69     __IN1(s) __IN2(s,s1,"w") : "=a" (_v) : "Nd" (port) ,##i ); return _v; } \
70     __IN1(s##_p) __IN2(s,s1,"w") __FULL_SLOW_DOWN_IO \
                               : "=a" (_v) : "Nd" (port) ,##i ); return _v; }
```

读者应该能推导出，根据 83 行的生成结果为：

```
unsigned char inb(unsigned short port) {
    unsigned char _v;
      __asm__ __volatile__ ("inb %w1, %0": "=a" (_v) : "Nd" (port),); return _v; }

unsigned char inb_p(unsigned short port) {
    unsigned char _v;
    __asm__ __volatile__ ("inb %w1, %0"
        "jmp 1f"
        "1: jmp 1f"
        "1:"
        : "=a" (_v) : "Nd" (port),);
    return _v; }
```

这些底层函数在本章要阅读的代码中经常要碰到。

最后还要说明，由于设备的多样性，设备驱动是一个需要整本专著的大课题，甚至一本专著还不够。例如，在 drivers 目录下的子目录 net、atm、isdn 等都是计算机网络论述方面的内容，其中的每一部分都是需要有专著加以介绍，而不是三言两语讲得清的。与其语焉不详地讲上几句，倒不如留待将来，或等有关专著的出现，所以我们在本书中干脆就不触及这些话题。此外，有些设备本身的原理和机制就很复杂（如显示设备），需要有专著加以介绍。我们既无足够的篇幅，也缺乏有关的专门知识来深入阐述这些设备的原理、机制和操作，而只能专注于它们的驱动程序。所以，我们在本书中只能集中在除网络设备（那是一个过于广阔的天地）以外的一些典型设备的驱动程序上。但是，我们相信读者在理解了这些内容以后就能举一反三，将基本的原理与实现方法推广到其他设备。另外，在内核的代码中，对同一种设备常常因为针对具体产品而开发了大同小异的驱动程序，例如对鼠标器就有好几种驱动程序。如果我们选择了某个具体产品的驱动程序作为实例，那只不过是它的代码适合用作实例，

或者我们碰巧选择了这些代码作为实例，而并不表示我们倾向于或偏爱该项具体的产品，更不表示我们倾向于生产该产品的厂商。

8.2　系统调用 mknod( )

“设备文件”是文件系统中代表设备的特殊文件。与普通的文件相比，设备文件在磁盘（或宿主文件系统所在的其他设备）上只占一个索引节点，而没有任何用于存放数据的记录块与之相联系。当然，这是因为设备文件的目的并不在于存储和读取数据，而只在于为应用程序提供一条通向具体设备的访问途径，使应用程序可以跟具体设备建立起连接。可想而知，既然没有用于数据的记录块，则索引节点中的记录块映象表（对于 Ext2 文件系统来说是 ext2_inode 结构中的数组 i_block[ ]）就没有什么用处了。所以，在这种情况下就用这个数组中的第一个元素，即 i_block[0]，记载目标设备的设备号。对于 Ext2 普通文件，这个数组中的元素在文件创建之初都是空的，只是在写文件时才随着记录块的分配而写入具体的内容（见“文件的读与写”）；而对于设备文件，则必须在创建时就将其所代表的设备号写入这个数组的第一个元素中。在“文件系统”一章中我们已经看到，普通文件（以及某些特殊文件）可以通过系统调用 open( )创建，只要在调用参数中将 O_CREAT 标志位设成 1，就可以让 open( )在目标文件不存在时先创建这个文件。此外，也可以通过系统调用 creat( )创建文件，事实上 sys_creat( )就是通过 sys_open( )实现的。可是，这两个系统调用都不能用来创建设备文件，因为设备文件的创建需要有一个参数来传递设备号，而系统调用 open( )和 creat( )的界面都不包括这个参数。实际上，从 Unix 的早期就为设备文件的创建另外设置了一个系统调用 mknod( )，并且一直沿用至今。

系统调用 mknod( )是通用的，可以用来创建任何类型的文件（除目录外），包括普通文件、特殊文件以及设备文件。不过，由于其他类型的文件大都有专用的系统调用，如普通文件可以用 open( )或 creat( )创建，FIFO 文件可以用 pipe( )创建，所以 mknod( )主要用于设备文件的创建。此外，不像 open( )和 creat( )那样集创建与打开于一身，mknod( )只是纯粹的创建。

在内核中，mknod( )是由 sys_mknod( )实现的，其代码在 fs/namei.c 中：

```
1205	asmlinkage long sys_mknod(const char * filename, int mode, dev_t dev)
1206	{
1207		int error = 0;
1208		char * tmp;
1209		struct dentry * dentry;
1210		struct nameidata nd;
1211
1212		if (S_ISDIR(mode))
1213			return -EPERM;
1214		tmp = getname(filename);
1215		if (IS_ERR(tmp))
1216			return PTR_ERR(tmp);
1217
1218		if (path_init(tmp, LOOKUP_PARENT, &nd))
1219			error = path_walk(tmp, &nd);
1220		if (error)
```

```
1221                goto out;
1222        dentry = lookup_create(&nd, 0);
1223        error = PTR_ERR(dentry);
1224        if (!IS_ERR(dentry)) {
1225            switch (mode & S_IFMT) {
1226            case 0: case S_IFREG:
1227                error = vfs_create(nd.dentry->d_inode,dentry,mode);
1228                break;
1229            case S_IFCHR: case S_IFBLK: case S_IFIFO: case S_IFSOCK:
1230                error = vfs_mknod(nd.dentry->d_inode,dentry,mode,dev);
1231                break;
1232            case S_IFDIR:
1233                error = -EPERM;
1234                break;
1235            default:
1236                error = -EINVAL;
1237            }
1238            dput(dentry);
1239        }
1240        up(&nd.dentry->d_inode->i_sem);
1241        path_release(&nd);
1242    out:
1243        putname(tmp);
1244
1245        return error;
1246    }
```

参数 dev 即为设备号，其类型为 dev_t，实际上是一个 16 位的无符号短整数，有关的定义见 include/asm_i386/posix_type.h 以及 include/linux/types.h：

```
10      typedef unsigned short  __kernel_dev_t;

14      typedef __kernel_dev_t        dev_t;
```

这个 16 位的无符号整数实际上分成两部分，其高 8 位为表示设备类别的“主设备号”，而低 8 位则为同一类别内的序号。30 年前，256 种不同的块设备，再加 256 种不同的字符设备，每种设备又可以有多至 256 台，似乎已是不可思议的了。但是，现在再来看，8 位的主设备号和次设备号的覆盖面和容量确实显得小了。可是，在用户界面上，也就是在系统调用界面上，一旦把设备号定义为 16 位无符号整数后就不能改变了，除非能找到一种与原来兼容的方案，否则就要引起兼容性问题。在系统调用界面已经定义的前提下，解决的办法无非就是设法既扩大容量又使新老界面兼容（显然很难），或者增添新的系统调用（现在没有）或新的机制（如前述的树状设备文件子系统），或者甚至三管齐下。但是，对于内核中的内部使用却不受用户界面的限制。所以，作为扩大设备号容量的准备，又在 include/linux/kdev.h 中定义了另一个数据类型 kdev_t：

```
67      typedef unsigned short kdev_t;
```

这个数据类型目前仍是 16 位的，但是其意图显然是在条件成熟时改成 32 位。

在 sys_mknod( )的代码中，只有 vfs_mknod( )是读者尚未看到过的。文件系统的 vfs 层上同类的函数有三个，即 vfs_create( )、vfs_mkdir( )以及 vfs_mknod( )。其中 vfs_create( )用于普通文件，vfs_mkdir( )用于目录节点，而 vfs_mknod( )则用于特殊文件，包括 FIFO、插口以及字符设备和块设备文件的创建。至于/proc 目录下的特殊文件，则一般是由内核生成，而不是由用户创建的，这一点读者在“文件系统”一章中已经看到过了。

函数 vfs_mknod( )的代码也在 fs/namei.c 中：

[sys_mknod( ) > vfs_mknod( )]

```
1176    int vfs_mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t dev)
1177    {
1178        int error = -EPERM;
1179
1180        mode &= ~current->fs->umask;
1181
1182        down(&dir->i_zombie);
1183        if ((S_ISCHR(mode) || S_ISBLK(mode)) && !capable(CAP_MKNOD))
1184            goto exit_lock;
1185
1186        error = may_create(dir, dentry);
1187        if (error)
1188            goto exit_lock;
1189
1190        error = -EPERM;
1191        if (!dir->i_op || !dir->i_op->mknod)
1192            goto exit_lock;
1193
1194        DQUOT_INIT(dir);
1195        lock_kernel( );
1196        error = dir->i_op->mknod(dir, dentry, mode, dev);
1197        unlock_kernel( );
1198    exit_lock:
1199        up(&dir->i_zombie);
1200        if (!error)
1201            inode_dir_notify(dir, DN_CREATE);
1202        return error;
1203    }
```

参数 dir 是一个 inode 结构指针，指向待创建设备文件的父节点，即其所在目录节点的 inode 结构，这是在 sys_mknod( )中通过 path_walk( )找到的。第二个参数 dentry，则指向代表着(如果已经存在)或将要代表待创建设备文件节点的目录项 dentry 数据结构。这个数据结构是在 sys_mknod( )中通过 lookup_create( )在内核中的 dentry 结构杂凑表中找到或者新创建的。

进入 vfs_mknod( )以后，首先是对权限的检验。以前讲过，每个进程的 fs_struct 结构中有个访问权限屏蔽位图 umask，每当一个进程要求创建文件时，就用这个位图将用户通过参数 mode 表达的“非分

之想”都过滤掉(1180 行)。对权限的检验分两步进行。第一步是 capable(CAP_MKNOD)，即检验当前进程是否允许创建设备节点，此项检验仅用于待创建节点为设备节点时（而不包括 FIFO 和插口节点）。第二步，即 may_create( )，则适用于所有节点，其代码读者已在“文件的打开与关闭”一节中见到过，此处就不重复了。这个函数中还包含了对目标节点是否业已存在的检验。显然，如果已经存在就不允许重复创建了。此外，由于我们在这里并不关心磁盘容量配额的问题，所以跳过 DQUOT_INIT( )。

创建设备节点的具体操作因文件系统而异，由具体文件系统通过其 inode_operations 结构中的函数指针 mknod 提供。对于 Ext2，这个函数是 ext2_mknod( )，其代码在 fs/ext2/namei.c 中：

[sys_mknod( ) > vfs_mknod( ) > ext2_mknod( )]

```
386	static int ext2_mknod (struct inode * dir, struct dentry *dentry,
					int mode, int rdev)
387	{
388		struct inode * inode = ext2_new_inode (dir, mode);
389		int err = PTR_ERR(inode);
390	
391		if (IS_ERR(inode))
392			return err;
393	
394		inode->i_uid = current->fsuid;
395		init_special_inode(inode, mode, rdev);
396		err = ext2_add_entry (dir, dentry->d_name.name, dentry->d_name.len,
397				inode);
398		if (err)
399			goto out_no_entry;
400		mark_inode_dirty(inode);
401		d_instantiate(dentry, inode);
402		return 0;
403	
404	out_no_entry:
405		inode->i_nlink--;
406		mark_inode_dirty(inode);
407		iput(inode);
408		return err;
409	}
```

首先通过 ext2_new_inode( )分配一个 inode 数据结构，这个函数的代码读者也已经看到过了。然后，就是设置这个数据结构，包括将设备号写入到该数据结构中，这是通过 init_special_inode( )完成的。实际上，设备节点（以及其他特殊文件节点）的特殊之处也正在于此。这个函数的代码在 fs/devices.c 中：

[sys_mknod( ) > vfs_mknod( ) > ext2_mknod( ) > init_special_inode( )]

```
200	void init_special_inode(struct inode *inode, umode_t mode, int rdev)
201	{
202		inode->i_mode = mode;
203		if (S_ISCHR(mode)) {
```

```
204             inode->i_fop = &def_chr_fops;
205             inode->i_rdev = to_kdev_t(rdev);
206         } else if (S_ISBLK(mode)) {
207             inode->i_fop = &def_blk_fops;
208             inode->i_rdev = to_kdev_t(rdev);
209             inode->i_bdev = bdget(rdev);
210         } else if (S_ISFIFO(mode))
211             inode->i_fop = &def_fifo_fops;
212         else if (S_ISSOCK(mode))
213             inode->i_fop = &bad_sock_fops;
214         else
215             printk(KERN_DEBUG"init_special_inode: bogus imode (%o)\n", mode);
216     }
```

这里主要是对两个结构成分的设置，一个是 file_operations 结构指针 i_fop，使它指向具体的数据结构 def_chr_fops 或 def_blk_fops；另一个是用来保存设备号的 i_rdev。注意在这里参数 rdev 的类型为 int(而不是 dev_t)，而 inode 结构中 i_rdev 的类型就是上述的 kdev_t。目前，kdev_t 仍是 16 位的，从用户进程传递下来的设备号也是 16 位的，如果把它看成 32 位整数的话，则其高 16 位总是 0。但是将来在系统调用 mknod( )的界面上可以把设备号的类型从 dev_t 改成 int，而根据其高 16 位是否为 0 确定所用的是 16 位设备号还是 32 位设备号，从而保持与原有界面兼容。同时，对 kdev_t 的定义也可以从 16 位改成 32 位。为了这个目的，内核代码中已经提供了一对 inline 函数 to_kdev_t( )和 kdev_t_to_nr( )，用来进行二者间的转换。但是目前这种转换只是从 16 位到 16 位，实际上不起什么作用，看一下 include/linux/kdevt_b 中的函数 to_kdev_t( )的代码就可以明白这一点：

```
87  static inline kdev_t to_kdev_t(int dev)
88  {
89    int major, minor;
90  #if 0
91    major = (dev >> 16);
92    if (!major) {
93        major = (dev >> 8);
94        minor = (dev & 0xff);
95    } else
96        minor = (dev & 0xffff);
97  #else
98    major = (dev >> 8);
99    minor = (dev & 0xff);
100 #endif
101   return MKDEV(major, minor);
102 }
```

这里条件编译“#if 0”下面这一块就是为将来准备的，读一下这部分代码就可以明白 32 位的设备号怎样与 16 位的设备号保持兼容。

还要提一下，inode 结构中有两个类型为 kdev_t 的字段。一个是 i_rdev，用于该索引节点所代表设备（例如/dev/tty0）的设备号；另一个是 i_dev，用于索引节点所在设备（例如/dev/hda1，假定目录/dev

在这个设备上）的设备号。

对于块设备，inode 结构中还有个指针 i_bdev，它指向一个 block_device 数据结构，那就是具体块设备的控制结构，定义于 include/linux/fs.h：

```
377     struct block_device {
378         struct list_head    bd_hash;
379         atomic_t            bd_count;
380     /*  struct address_space    bd_data; */
381         dev_t               bd_dev;  /* not a kdev_t - it's a search key */
382         atomic_t            bd_openers;
383         const struct block_device_operations *bd_op;
384         struct semaphore    bd_sem;  /* open/close mutex */
385     };
```

里面有个指针 bd_op 指向一个 block_device_operations 结构，那就是这个设备的函数跳转表。内核中为块设备设置了一个杂凑表 bdev_hashtable[ ]，每当首次创建或打开某个块设备文件时就为其创建一个 block_device 结构，并且根据设备号的杂凑值将此结构通过队列头 bd_hash 挂入杂凑表的某个队列中，以便查找。

在 init_special_inode( )中，通过函数 bdget( )根据设备号找到或者创建给定设备的 block_device 数据结构，其代码在 fs/block_dev.c 中：

[sys_mknod( ) > vfs_mknod( ) > ext2_mknod( ) > init_special_inode( ) > bdget( )]

```
429     struct block_device *bdget(dev_t dev)
430     {
431         struct list_head * head = bdev_hashtable + hash(dev);
432         struct block_device *bdev, *new_bdev;
433         spin_lock(&bdev_lock);
434         bdev = bdfind(dev, head);
435         spin_unlock(&bdev_lock);
436         if (bdev)
437             return bdev;
438         new_bdev = alloc_bdev( );
439         if (!new_bdev)
440             return NULL;
441         atomic_set(&new_bdev->bd_count, 1);
442         new_bdev->bd_dev = dev;
443         new_bdev->bd_op = NULL;
444         spin_lock(&bdev_lock);
445         bdev = bdfind(dev, head);
446         if (!bdev) {
447             list_add(&new_bdev->bd_hash, head);
448             spin_unlock(&bdev_lock);
449             return new_bdev;
450         }
451         spin_unlock(&bdev_lock);
```

```
452         destroy_bdev(new_bdev);
453         return bdev;
454     }
```

首先是试着在杂凑表 bdev_hashtable[ ]里的相应队列中通过 bdfind( )寻找，如果找到就完成了（见437行）。要是找不到，那就要通过 alloc_bdev( )分配一个数据结构。由于在 alloc_bdev( )的过程中当前进程有可能进入睡眠，所以在分配到了以后还要再调用 bdfind( )寻找一遍，以防因别的进程抢先为同一设备分配了数据结构而造成冲突。在确认并未造成冲突以后，就将该数据结构挂入相应的杂凑队列中。值得注意的是，这里只是将数据结构中的指针 bd_op 初始化成 NULL，而将这个指针的落实留待第一次打开这个设备文件的时候。读者在"文件系统的安装与拆卸"一节中曾看到，在安装文件系统时通过 get_blkfops( )根据设备的主设备号取得指向其 block_device_operations 结构的指针，就是因为在安装时隐含着打开块设备文件的操作。

回到 ext2_mknod( )的代码中，ext2_add_entry( )将新创建的节点加进所在目录在磁盘上的目录文件中，mark_inode_dirty( )将新创建的 inode 结构设置成"脏"，而 d_instantiate( )则使内存中的目录项 dentry 结构与 inode 结构间互相挂上钩，这些函数读者都已不是第一次看到了。由于新创建的 inode 结构设置成了"脏"，内核在"同步"内存中的 inode 结构与磁盘上的索引节点的时候，就会将这个 inode 结构的内容写到磁盘上分配给这个文件的索引节点，即 ext2_inode 数据结构中。由于 ext2_inode 结构中并不存在 i_rdev 这么个成分，而对于设备文件却又不需要使用 i_block[ ]数组，所以就挪用其 i_block[0]来保存设备号。要了解这一点，我们不妨看一下文件 fs/ext2/inode.c 中函数 ext2_update_inode( )代码的一个片段：

```
1211        if (S_ISCHR(inode->i_mode) || S_ISBLK(inode->i_mode))
1212            raw_inode->i_block[0]=cpu_to_le32(kdev_t_to_nr(inode->i_rdev));
1213        else for (block = 0; block < EXT2_N_BLOCKS; block++)
1214            raw_inode->i_block[block] = inode->u.ext2_i.i_data[block];
```

这里的 raw_inode 是一个 ext2_inode 结构指针，而 inode 是一个 inode 结构指针。这二者一个是存储在磁盘上的索引节点，另一个是索引节点在内存中的表现，但是二者在内容上并不完全相同，inode 结构中含有许多动态的信息。如果 inode 结构代表着一个字符设备或块设备，则相应 ext2_inode 结构中的 i_block[0]用于存放设备号，其内容来自 inode 结构中的 i_rdev（须转换成"Little Ending"）。要不然，则 i_block[ ]数组中各个元素的内容来自 inode 结构中 ext2_inode_info 部分的 i_data[ ]数组的相应元素，那是一些直接或间接地指向各个数据记录块的指针。

反过来，当通过 ext2_read_inode( )从磁盘上读入索引节点，并为之在内存中创建相应的 inode 结构时，则先将 i_block[ ]数组全部复制到 i_data[ ]数组中，然后如果是设备文件就调用 init_special_inode( )将 i_block[0]的内容（从"Little Ending"转换回 CPU 所采用的制式）填入 inode 结构中的 i_rdev。以下是 ext2_read_inode( )中有关的片断（见 fs/ext2/inode.c）：

```
1052        /*
1053         * NOTE! The in-memory inode i_data array is in little-endian order
1054         * even on big-endian machines: we do NOT byteswap the block numbers!
1055         */
```

```
1056        for (block = 0; block < EXT2_N_BLOCKS; block++)
1057            inode->u.ext2_i.i_data[block] = raw_inode->i_block[block];
1058
1059        if (inode->i_ino == EXT2_ACL_IDX_INO ||
1060            inode->i_ino == EXT2_ACL_DATA_INO)
1061            /* Nothing to do */ ;
1062        else if (S_ISREG(inode->i_mode)) {
            ......
1066        } else if (S_ISDIR(inode->i_mode)) {
            ......
1069        } else if (S_ISLNK(inode->i_mode)) {
            ......
1076        } else
1077            init_special_inode(inode, inode->i_mode,
1078                               le32_to_cpu(raw_inode->i_block[0]));
```

这就又回到了上面的 init_special_inode( )。

对于设备文件，将 ext2_inode 结构中的 i_block[ ]数组复制到 inode 结构中的 i_data[ ]数组实际上是没有意义的。至于 inode 结构中的 block_device 结构指针 i_bdev，则本来就是动态的，所以根本就不保存在磁盘上的索引节点中。

8.3 可安装模块

早在第 1 章我们就曾提到，Linux 采用的是整体式的内核结构（monolithic kernel），这种结构的内核一般不能动态地增加新的功能。为此，Linux 提供了一种全新的机制，叫(可安装)“模块”（module）。利用这个机制，可以根据需要，在不必对内核重新编译连接的条件下，将可安装模块动态地插入运行中的内核，成为内核的一个有机组成部分；或者从内核移走已经安装的模块。正是这种机制，使得内核的内存映象保持最小，但却具有很大的灵活性和可扩充性。

可安装模块是可以在系统运行时动态地安装和拆卸的内核软件。严格说来，这种软件的作用并不限于设备驱动，例如有些文件系统就是以可安装模块的形式实现的。但是，另一方面，它主要用来实现设备驱动程序或者与设备驱动密切相关的部分（如文件系统），所以我们将它放在本章中介绍。

在应用程序界面上，内核通过 4 个系统调用支持可安装模块的动态安装和拆卸，它们是 create_module( )、init_module( )、query_module( )以及 delete_module( )。通常，用户一般不需要直接跟这些系统调用打交道，而可以用系统提供的工具/sbin/insmod（插入模块）和/sbin/rmmod（移走模块）来安装和拆卸可安装模块。当然，这两个工具最终还是要通过这些系统调用完成有关操作。大体上说，/sbin/insmod 所做的事情有这么一些：

- 打开待安装模块并将其读入到用户空间。所谓“模块”就是经过编译但未经连接的.o 文件。
- 模块中必定有一些在模块内部无法落实的符号（函数名或变量名），对这些符号的引用必须连接到内核中的相应符号，也就是必须把这些符号在内核映象中的地址填入模块中需要访问这些符号的指令中以及数据结构中。为此目的，需要通过系统调用 query_module( )向内核询问这些符号在内核中的地址。如果内核允许“移出”这些符号的地址，就会返回有关的“符号

表”。有些符号可能并不属于内核本身，而属于已经安装的其他模块。

- 取得了内核“移出”的符号表以后，就应该可以使模块中所有的符号引用都得到落实了。这部分操作与编译后的连接相似。不过，常规的“连接”常常是双向的，而现在只是在模块中引用内核里的符号或者某些已经安装的模块中的符号，而内核却并不要求(也不能)反过来引用这个待安装模块中的符号。当然，内核最后一定会要访问模块中的某些变量或调用模块中的某些函数，否则模块中的函数就得不到执行，模块的存在也就失去了意义。从这个意义上说，模块与内核的连接只完成了一半，我们不妨称之为“完成了单向连接”的模块映象。
- 然后，通过系统调用 create_module( )在内核中创建一个 module 数据结构，并且“预订”所需的系统（内核）空间。
- 最后，通过系统调用 init_module( )把用户空间中完成了单向连接的模块映象装入内核空间，再调用模块中一个名为 init_module( )的函数。注意，不要把可安装模块中的函数 init_module( )与系统调用 init_module( )搞混淆了，这完全是两码事。系统调用 init_module( )在内核中的实现是 sys_init_module( )，这是由内核提供的，整个内核中只有这么一个。而模块中的函数 init_module( )，则是由可安装模块本身提供的，每个模块都有这样一个函数。通常，每个模块的 init_module( )负责向内核“登记”本模块中的一些包含着函数指针的数据结构（例如 file_operations 结构）。完成了这种登记以后，模块与内核之间的连接（另一个方向上的连接）才真正完成了。

顾名思义，系统调用 delete_module( )将模块的 module 结构释放，并且将模块映象所占的内核空间释放。此外，还有一件重要的事情就是调用模块中一个名为 cleanup_module( )的函数。与 init_module( )函数一样，每个可安装模块都有其自己的 cleanup_module( )函数。通常，在这个函数中要向内核撤销对由本模块提供的数据结构（从而函数指针）的登记，使内核在模块拆卸以后不至于再企图访问这些数据结构（以及函数指针）。

下面，我们先看 4 个系统调用的代码，然后再通过一个实际的模块来看一下典型的 init_module( )函数和 cleanup_module( )函数。函数 sys_create_module( )、sys_init_module( )、sys_query_module( )，还有 sys_delete_module( )的代码都在 kernel/module.c 中。

先看 sys_query_module( )。

```
874    asmlinkage long
875    sys_query_module(const char *name_user, int which, char *buf, size_t bufsize,
876                size_t *ret)
877    {
878        struct module *mod;
879        int err;
880
881        lock_kernel( );
882        if (name_user == NULL)
883            mod = &kernel_module;
884        else {
885            long namelen;
886            char *name;
887
```

```
888             if ((namelen = get_mod_name(name_user, &name)) < 0) {
889                 err = namelen;
890                 goto out;
891             }
892             err = -ENOENT;
893             if (namelen == 0)
894                 mod = &kernel_module;
895             else if ((mod = find_module(name)) == NULL) {
896                 put_mod_name(name);
897                 goto out;
898             }
899             put_mod_name(name);
900         }
901
902         switch (which)
903         {
904         case 0:
905             err = 0;
906             break;
907         case QM_MODULES:
908             err = qm_modules(buf, bufsize, ret);
909             break;
910         case QM_DEPS:
911             err = qm_deps(mod, buf, bufsize, ret);
912             break;
913         case QM_REFS:
914             err = qm_refs(mod, buf, bufsize, ret);
915             break;
916         case QM_SYMBOLS:
917             err = qm_symbols(mod, buf, bufsize, ret);
918             break;
919         case QM_INFO:
920             err = qm_info(mod, buf, bufsize, ret);
921             break;
922         default:
923             err = -EINVAL;
924             break;
925         }
926     out:
927         unlock_kernel( );
928         return err;
929     }
```

参数 name_user 为查询对象所在模块的模块名，如果这个指针为 0 即指内核本身。参数 which 表示查询的内容，如 QM_SYMBOLS 表示查询目标模块的符号表，QM_MODULES 表示查询内核中已安装模块的清单，而 QM_DEPS 则查询给定模块对其他模块的依赖（如果模块 A 引用模块 B 中的符号，则 A 依赖于 B）。查询的结果通过缓冲区 buf 返回。

每个已安装模块在内核中都有一个 module 数据结构（通过系统调用 create_module( )创建），include/linux/module.h 中定义了 module 以及一些有关的数据结构：

```
37    struct module_symbol
38    {
39        unsigned long value;
40        const char *name;
41    };
42
43    struct module_ref
44    {
45        struct module *dep; /* "parent" pointer */
46        struct module *ref; /* "child" pointer */
47        struct module_ref *next_ref;
48    };
49
50    /* TBD */
51    struct module_persist;
52
53    struct module
54    {
55        unsigned long size_of_struct;   /* == sizeof(module) */
56        struct module *next;
57        const char *name;
58        unsigned long size;
59
60        union
61        {
62            atomic_t usecount;
63            long pad;
64        } uc;                /* Needs to keep its size - so says rth */
65
66        unsigned long flags;        /* AUTOCLEAN et al */
67
68        unsigned nsyms;
69        unsigned ndeps;
70
71        struct module_symbol *syms;
72        struct module_ref *deps;
73        struct module_ref *refs;
74        int (*init)(void);
75        void (*cleanup)(void);
76        const struct exception_table_entry *ex_table_start;
77        const struct exception_table_entry *ex_table_end;
78    #ifdef __alpha__
79        unsigned long gp;
80    #endif
```

```
81         /* Members past this point are extensions to the basic
82            module support and are optional.  Use mod_member_present( )
83            to examine them.  */
84         const struct module persist *persist_start;
85         const struct module_persist *persist_end;
86         int (*can_unload)(void);
87         int runsize;            /* In modutils, not currently used */
88         const char *kallsyms_start; /* All symbols for kernel debugging */
89         const char *kallsyms_end;
90         const char *archdata_start; /* arch specific data for module */
91         const char *archdata end;
92         const char *kernel_data;    /* Reserved for kernel internal use */
93     };
```

这里主要定义了三种数据结构，第一种数据结构是 module_symbol，每个 module_symbol 结构描述了一个符号，包括符号名及其所在的地址，或者说符号的值。第二种是 module_ref，用来描述模块间的依赖关系。最后就是 module 结构。内核中的 module 结构通过它们的指针 next 连成一个链。每个模块都有个模块名 name，size 则为模块的大小，函数指针 init 和 cleanup 分别指向模块的 init_module( ) 和 cleanup_module( )函数。指针 syms 指向一个 module_symbol 结构数组，而 nsyms 指明了数组的大小，此即为模块的符号表。对模块中的具体符号是否放在其符号表中也是可以选择的，对此我们还将详细介绍。指针 deps 指向一个 module_ref 结构数组，ndeps 则为该数组的大小。这个数组中的每一个元素都通过其指针 dep 指向一个 module 结构，这些就是给定模块所依赖的模块。要安装一个模块时，这个模块所依赖的所有模块都必须已经安装好。与依赖关系（dep）方向相反的是“被引用”关系 ref。就是说，如果模块 A 引用了模块 B 中的符号，则 A 依赖于 B，或者说 B 被 A 引用。注意这里的所谓“依赖”，即“dep”，实际上就是“引用”，只不过是主动语态的“引用”，而“ref”则是被动语态的“引用”，所以二者方向相反。

如果 A 依赖于 B，则模块 A 的 deps 数组中有一个元素指向模块 B。但是，光有这样的联系还不够方便，假如某一个应用程序要拆卸模块 B，那就要扫描所有已安装模块的 deps 数组才能知道是否还有模块依赖于 B（如果有，则 B 还有“用户”而不能拆除）。所以，在 module 结构中还有个指针 refs，它指向一个 module_ref 结构链，链中的每个结构都通过指针 ref 指向一个依赖于它的模块，并且就在那个模块的 deps 数组中。在我们这个例子中，模块 A 的 deps 数组中就应该有这么一个 module_ref 结构，其指针 dep 指向 B；指针 ref 则指向 A 自身，并且通过链接指针 next_ref 链入 B 的 refs 链中。这样，从模块 A 出发，通过其 deps 数组就能找到 A 所依赖的所有模块，包括 B 在内。反过来，从模块 B 出发，则通过其 refs 链就能找到所有引用了 B 的模块，包括 A 在内。这里，“依赖”关系是静态的，一个模块依赖于哪几个模块是固定的，模块一经实现，这一点就定下来了，所以适合于使用数组。而反过来，“被引用”关系则是动态的，一个模块被几个模块引用随时都可能变动，所以要采用动态的 module_ref 结构链（即队列）。

虽然内核并不是可安装模块，但是它也有符号表，也受到其他模块的引用，将其看成可安装模块可以简化程序设计，所以为内核也定义了一个 module 结构，从 kernel_module 开始，所有已安装模块的 module 结构都链接在一起成为一条链，内核中的全局量 module_list 就指向这条链。

此处建议读者先不忙往下看，先把前面介绍的三个数据结构以及相互之间的联系整理一下，画张

图。这对理解模块机制源代码会有好处。

理解了这几个数据结构，sys_query_module( )的代码就很简单了。如前所述，如果参数 name_user 为 NULL 就表示查询的对象为内核本身，其 module 结构即 kernel_module，这是在 kernel/module.c 中定义的：

```
41      static struct module kernel_module =
42      {
43          size_of_struct:      sizeof(struct module),
44          name:                "",
45          uc:                  {ATOMIC_INIT(1)},
46          flags:               MOD_RUNNING,
47          syms:                __start___ksymtab,
48          ex_table_start:      __start___ex_table,
49          ex_table_end:        __stop___ex_table,
50          kallsyms_start:      __start___kallsyms,
51          kallsyms_end:        __stop___kallsyms,
52      };
```

凡是不出现在初始值定义中的字段（如 deps 和 refs 等等），其值均为 0 或 NULL。显然，内核没有 init_module( )和 cleanup_module( )这两个函数，因为相应的函数指针都是 NULL。同时，内核没有 deps 数组，开始时也没有 refs 链。可是，这个数据结构中的指针 syms 指向__start___ksymtab，这就是内核符号表的起始地址。符号表的大小 nsyms 为 0，但是在系统初始化时会在函数 init_modules( )中将其设置成正确的数值。

如果参数 name_user 不是 NULL，那就是查询一个特定的模块了。所以在通过 get_mod_name( )从用户空间复制模块名以后就通过 find_module( )在队列 module_list 中寻找相应的 module 结构。这个函数的代码也在 kernel/module.c 中：

[sys_query_module( ) > find_module( )]

```
993     /*
994      * Look for a module by name, ignoring modules marked for deletion.
995      */
996
997     struct module *
998     find_module(const char *name)
999     {
1000        struct module *mod;
1001
1002        for (mod = module_list; mod ; mod = mod->next) {
1003            if (mod->flags & MOD_DELETED)
1004                continue;
1005            if (!strcmp(mod->name, name))
1006                break;
1007        }
1008
```

```
1009        return mod;
1010    }
```

至于对具体查询的服务，即 qm_modules( )、qm_symbols ( )、qm_deps( )等等，则很简单了。我们看一下 kernel/module.c 中函数 qm_deps( )的代码:

[sys_query_module( ) > qm_deps( )]

```
701    static int
702    qm_deps(struct module *mod, char *buf, size_t bufsize, size_t *ret)
703    {
704        size_t i, space, len;
705
706        if (mod == &kernel_module)
707            return -EINVAL;
708        if (!MOD_CAN_QUERY(mod))
709            if (put_user(0, ret))
710                return -EFAULT;
711            else
712                return 0;
713
714        space = 0;
715        for (i = 0; i < mod->ndeps; ++i) {
716            const char *dep_name = mod->deps[i].dep->name;
717
718            len = strlen(dep_name)+1;
719            if (len > bufsize)
720                goto calc_space_needed;
721            if (copy_to_user(buf, dep_name, len))
722                return -EFAULT;
723            buf += len;
724            bufsize -= len;
725            space += len;
726        }
727
728        if (put_user(i, ret))
729            return -EFAULT;
730        else
731            return 0;
732
733    calc_space_needed:
734        space += len;
735        while (++i < mod->ndeps)
736            space += strlen(mod->deps[i].dep->name)+1;
737
738        if (put_user(space, ret))
739            return -EFAULT;
```

```
740         else
741             return -ENOSPC;
742     }
```

系统调用返回 0 表示成功，此时参数 ret 所指的整数含有所依赖模块的个数，而 buf 中则为这些模块的模块名。否则表示执行失败，出错代码为 ENOSPC 表示 buf 的空间太小，此时 ret 所指的整数含有所需的空间大小。这些代码就不需要解释了，其他几个也是一样，有兴趣的读者可以自己阅读。

再看系统调用 create_module( )的实现 sys_create_module( )，也在 kernel/module.c 中：

```
276     /*
277      * Allocate space for a module.
278      */
279
280     asmlinkage unsigned long
281     sys_create_module(const char *name_user, size_t size)
282     {
283         char *name;
284         long namelen, error;
285         struct module *mod;
286
287         if (!capable(CAP_SYS_MODULE))
288             return -EPERM;
289         lock_kernel( );
290         if ((namelen = get_mod_name(name_user, &name)) < 0) {
291             error = namelen;
292             goto err0;
293         }
294         if (size < sizeof(struct module)+namelen) {
295             error = -EINVAL;
296             goto err1;
297         }
298         if (find_module(name) != NULL) {
299             error = -EEXIST;
300             goto err1;
301         }
302         if ((mod = (struct module *)module_map(size)) == NULL) {
303             error = -ENOMEM;
304             goto err1;
305         }
306
307         memset(mod, 0, sizeof(*mod));
308         mod->size_of_struct = sizeof(*mod);
309         mod->next = module_list;
310         mod->name = (char *)(mod + 1);
311         mod->size = size;
312         memcpy((char*)(mod+1), name, namelen+1);
```

```
313
314         put_mod_name(name);
315
316         module_list = mod;  /* link it in */
317
318         error = (long) mod;
319         goto err0;
320     err1:
321         put_mod_name(name);
322     err0:
323         unlock_kernel( );
324         return error;
325     }
```

只有特权用户才允许在内核中创建模块，capable(CAP_SYS_MODULE)就是对当前进程是否有这种特权的检查。参数 size 表示模块的大小，包括模块的映象本身以及一个 module 结构的大小，再加上模块名的长度。显然，size 至少也得等于后两者的和，否则就肯定错了。模块是以模块名来惟一地加以标识的，所以先要通过 find_module( )检查，看是否已经有同名的模块存在。通过了所有这些检验以后，就调用 module_map( )分配空间。对于 i386 结构的 CPU，module_map( )在 include/asm_i386/pgtable.h 中定义为 vmalloc( ):

```
7     #define module_map(x)        vmalloc(x)
```

分配得的空间的开头用作模块的 module 数据结构（见 307 行），然后是模块名字符串（见 312 行）。注意这里（mod+1）表示从 mod 所指的地址加上一个 module 结构大小的地方。最后剩下的用于模块的映象。显然，新建立的 module 结构基本上是空的，其内容有待于从用户空间传递过来。

对模块的 module 结构加以初始化，并将其链入 module_list 前端（见 309 行和 316 行）。此后，创建模块的操作就完成了，系统调用返回内核中为这个模块所分配的空间地址。下一步是系统调用 init_module( )。

如前所述，在系统调用 init_module( )之前要由应用程序在用户空间完成模块与内核符号间的连接，这个过程与“连接”，即 ld 的过程相似。由于这是在用户空间完成的，不属于内核的活动，所以我们不深入到这个过程中去了，有兴趣的读者可以从 GNU 网站下载 insmod 的源程序自己阅读。

相比之下，sys_init_module( )的代码（见 kernel/module.c）就比较大了，我们分段阅读。

[sys_init_module( )]

```
327     /*
328      * Initialize a module.
329      */
330
331     asmlinkage long
332     sys_init_module(const char *name_user, struct module *mod_user)
333     {
```

```
334        struct module mod_tmp, *mod;
335        char *name, *n_name, *name_tmp = NULL;
336        long namelen, n_namelen, i, error;
337        unsigned long mod_user_size;
338        struct module_ref *dep;
339
340        if (!capable(CAP_SYS_MODULE))
341            return -EPERM;
342        lock_kernel( );
343        if ((namelen = get_mod_name(name_user, &name)) < 0) {
344            error = namelen;
345            goto err0;
346        }
347        if ((mod = find_module(name)) == NULL) {
348            error = -ENOENT;
349            goto err1;
350        }
351
352        /* Check module header size.  We allow a bit of slop over the
353           size we are familiar with to cope with a version of insmod
354           for a newer kernel.  But don't over do it. */
355        if ((error = get_user(mod_user_size, &mod_user->size_of_struct)) != 0)
356            goto err1;
357        if (mod_user_size < (unsigned long)&((struct module *)0L)->persist_start
358            || mod_user_size > sizeof(struct module) + 16*sizeof(void*)) {
359            printk(KERN_ERR "init_module: Invalid module header size.\n"
360                   KERN_ERR "A new version of the modutils is likely "
361                    "needed.\n");
362            error = -EINVAL;
363            goto err1;
364        }
365
366        /* Hold the current contents while we play with the user's idea
367           of righteousness.  */
368        mod_tmp = *mod;
369        name_tmp = kmalloc(strlen(mod->name) + 1, GFP_KERNEL);
                                                      /* Where's kstrdup( )? */
370        if (name_tmp == NULL) {
371            error = -ENOMEM;
372            goto err1;
373        }
374        strcpy(name_tmp, mod->name);
375
376        error = copy_from_user(mod, mod_user, mod_user_size);
377        if (error) {
378            error = -EFAULT;
379            goto err2;
380        }
```

```
381
382         /* Sanity check the size of the module.  */
383         error = -EINVAL;
384
385         if (mod->size > mod_tmp.size) {
386             printk(KERN_ERR "init_module: Size of initialized module "
387                     "exceeds size of created module.\n");
388             goto err2;
389         }
390
```

首先还是对权限的检验。接着就是从用户空间复制模块名，并通过 find_module( )从 module_list 找到目标模块的 module 数据结构。在系统调用 init_module( )之前，应用程序要在用户空间为目标模块准备一个 module 数据结构，并且将指向这个结构的指针作为参数传给内核，这就是这里的指针 mod_user。此时内核中虽然已经由 sys_create_module( )创建了目标模块的 module 数据结构，但是这个数据结构基本上还是空的，其内容只能来自用户空间。现在，就要把用户空间 module 结构的内容复制到内核中的对应 module 结构中。可是，由于版本等等的原因，用户空间的 module 结构有可能与内核中的不完全一样。为了防止因用户空间的 module 结构与内核中的 module 结构大小不符而造成麻烦，先把用户空间 module 结构中的 size_of_struct 字段复制过来加以检查。在 module 数据结构的定义中，从 persist_start 开始的三个指针是内核对这个数据结构的扩充，用户空间的 module 结构有可能并不包括这些字段（读者不妨试一下“man init_module”看看），所以检查其大小时的条件之一是不小于结构中 persist_start 以前的那一部分。同时，这个大小也不能超过内核中 module 结构的大小加上 16 个指针，即 64 个字节的位置。通过了对结构大小的检查以后，先把内核中的 module 结构暂时保存在堆栈中作为后备（见 368 行），然后就从用户空间复制其 module 结构。复制时是以内存中的 module 结构大小为准的，以免“冲坏”内核中的内存空间。复制过来以后，还要核对“新版”数据结构中指明的模块大小与原来“预约”的模块大小是否一致（见 385 行）。

通过了对结构大小的检验以后，下一步就是检查结构中内容的合理性。继续往下看：

[sys_init_module( )]

```
391         /* Make sure all interesting pointers are sane.  */
392
393         if (!mod_bound(mod->name, namelen, mod)) {
394             printk(KERN_ERR "init_module: mod->name out of bounds.\n");
395             goto err2;
396         }
397         if (mod->nsyms && !mod_bound(mod->syms, mod->nsyms, mod)) {
398             printk(KERN_ERR "init_module: mod->syms out of bounds.\n");
399             goto err2;
400         }
401         if (mod->ndeps && !mod_bound(mod->deps, mod->ndeps, mod)) {
402             printk(KERN_ERR "init_module: mod->deps out of bounds.\n");
403             goto err2;
404         }
```

```
405        if (mod->init && !mod bound(mod->init, 0, mod)) {
406            printk(KERN_ERR "init_module: mod->init out of bounds.\n");
407            goto err2;
408        }
409        if (mod->cleanup && !mod_bound(mod->cleanup, 0, mod)) {
410            printk(KERN_ERR "init_module: mod->cleanup out of bounds.\n");
411            goto err2;
412        }
413        if (mod->ex_table start > mod->ex_table_end
414            || (mod->ex table_start &&
415            !((unsigned long)mod->ex_table_start >=
                                    ((unsigned long)mod + mod->size_of_struct)
416              && ((unsigned long)mod->ex_table_end
417                  < (unsigned long)mod + mod->size)))
418            || (((unsigned long)mod->ex table_start
419             - (unsigned long)mod->ex table_end)
420            % sizeof(struct exception_table entry))) {
421            printk(KERN ERR "init_module: mod >ex table_* invalid.\n");
422            goto err2;
423        }
424        if (mod->flags & ~MOD_AUTOCLEAN) {
425            printk(KERN_ERR "init_module: mod->flags invalid.\n");
426            goto err2;
427        }
428    #ifdef __alpha
429        if (!mod_bound(mod->gp - 0x8000, 0, mod)) {
430            printk(KERN_ERR "init_module: mod->gp out of bounds.\n");
431            goto err2;
432        }
433    #endif
434        if (mod_member_present(mod, can unload)
435            && mod->can_unload && !mod_bound(mod->can_unload, 0, mod)) {
436            printk(KERN_ERR "init_module: mod->can_unload out of bounds.\n");
437            goto err2;
438        }
439        if (mod_member_present(mod, kallsyms_end)) {
440            if (mod->kallsyms end &&
441            (!mod_bound(mod->kallsyms start, 0, mod) ||
442             !mod_bound(mod->kallsyms end, 0, mod))) {
443            printk(KERN_ERR "init_module: mod->kallsyms out of bounds.\n");
444            goto err2;
445            }
446            if (mod >kallsyms_start > mod->kallsyms_end) {
447            printk(KERN ERR "init_module: mod >kallsyms invalid.\n");
448            goto err2;
449            }
450        }
451        if (mod_member_present(mod, archdata_end)) {
```

```
452             if (mod->archdata_end &&
453             (!mod_bound(mod->archdata_start, 0, mod) ||
454              !mod_bound(mod->archdata_end, 0, mod))) {
455             printk(KERN_ERR "init_module: mod->archdata out of bounds.\n");
456             goto err2;
457             }
458             if (mod->archdata_start > mod->archdata_end) {
459             printk(KERN_ERR "init_module: mod->archdata invalid.\n");
460             goto err2;
461             }
462         }
463         if (mod_member_present(mod, kernel_data) && mod->kernel_data) {
464             printk(KERN_ERR "init_module: mod->kernel_data must be zero.\n");
465             goto err2;
466         }
467
468         /* Check that the user isn't doing something silly with the name.  */
469
470         if ((n_namelen = get_mod_name(mod->name - (unsigned long)mod
471                           + (unsigned long)mod_user,
472                           &n_name)) < 0) {
473             printk(KERN_ERR "init_module: get_mod_name failure.\n");
474             error = n_namelen;
475             goto err2;
476         }
477         if (namelen != n_namelen || strcmp(n_name, mod_tmp.name) != 0) {
478             printk(KERN_ERR "init_module: changed module name to "
479                     "`%s' from `%s'\n",
480                    n_name, mod_tmp.name);
481             goto err3;
482         }
483
```

这里的宏操作 mod_bound( )用来检查由用户提供的指针所指的对象是否落在模块的边界内，定义于 include/linux/module.h。

```
133     /* Check if an address p with number of entries n is within
                                                    the body of module m */
134     #define mod_bound(p, n, m) \
        ((unsigned long)(p) >= ((unsigned long)(m) + ((m)->size_of_struct)) && \
135                  (unsigned long)((p)+(n)) <= (unsigned long)(m) + (m)->size)
```

以第 393 行对模块名的检验为例，要检查的是 module 结构中的指针 name 指向为该模块分配的缓冲区内部，但是又不落在 module 结构的内部，同时，长度为 namelen 的字符串又不越出模块缓冲区的边界。简言之，这个字符串从必须在模块映象的范围内。类似地，如果模块的符号表以及 deps 数组非空，则也必须落在为模块分配的缓冲区中、module 结构外。不光是对指向数据结构的指针，对函数指

针也是一样，init_module( )和 cleanup_module( )两个函数必须在模块映象的范围中。

在模块映象中也可以包含对异常的处理。发生于一些特殊地址上的异常，可以通过一种描述结构 exception_table_entry 规定对异常的反应和处理，这些描述结构在可执行映象连接时都被集中在一个数组中（内核的 exception_table_entry 结构数组为__start___ex_table[ ]）。当异常发生时，内核的异常响应程序会先搜索这个数组，看看是否对所发生的异常（根据异常发生时的地址）规定了特殊的处理。我们在第 3 章中曾讲到过有关的情景，读者可以回过去看一下。当内核支持可安装模块时，其异常响应程序会扫描 module_list，对于已安装的每个模块都搜索其异常处理描述表（如前所述，内核本身也被看作是一个模块）。理所当然，每个模块的异常处理描述表也必须在模块映象的范围内。虽然对异常处理描述表的检验没有使用宏操作 bound，但是其精神是一致的。之所以在这里不使用 bound，是因为异常处理描述表是以起点和终点而不是起点加长度来界定其范围的。

内核中 module 结构内的最后一个字段是函数指针 can_unload，但是这属于 module 结构的扩充部分，用户空间的 module 结构可能包括也可能不包括这个字段。如果包括了这个字段，那就也要保证这个指针指向模块的映象内部。那么，怎样才能知道用户空间的 module 结构是否包括这个字段呢？方法是检查这个结构的大小，为此目的定义了一个宏操作 mod_member_present，它是在 module.h 中定义的：

```
125    /* When struct module is extended, we must test whether the new member
126       is present in the header received from insmod before we can use it.
127       This function returns true if the member is present.  */
128
129    #define mod_member_present(mod,member)                    \
130        ((unsigned long)(&((struct module *)0L)->member + 1)        \
131         <= (mod)->size_of_struct)
```

最后，对于模块名还要作一番检验。虽然在前面已经根据参数 name_user 从用户空间复制了作为系统调用参数的模块名，但是这个模块名是否与用户空间 module 结构中所指示的模块名一致呢？显然，不能排除不一致的可能性，所以现在要根据 module 结构的内容把模块映象中的模块名也复制过来，再与原先使用的模块名比较（见 470 至 477 行）。

经过了所有这些检验以后，可以从用户空间把模块的映象复制过来了。我们往下看：

[sys_init_module( )]

```
484        /* Ok, that's about all the sanity we can stomach; copy the rest.  */
485
486        if (copy_from_user((char *)mod+mod_user_size,
487                    (char *)mod_user+mod_user_size,
488                    mod->size-mod_user_size)) {
489            error = -EFAULT;
490            goto err3;
491        }
492
493        if (module_arch_init(mod))
494            goto err3;
495
```

```
496         /* On some machines it is necessary to do something here
497            to make the I and D caches consistent.  */
498         flush_icache_range((unsigned long)mod, (unsigned long)mod + mod->size);
499 
500         mod->next = mod_tmp.next;
501         mod->refs = NULL;
502 
503         /* Sanity check the module's dependents */
504         for (i = 0, dep = mod->deps; i < mod->ndeps; ++i, ++dep) {
505             struct module *o, *d = dep->dep;
506 
507             /* Make sure the indicated dependencies are really modules.  */
508             if (d == mod) {
509                 printk(KERN_ERR "init_module: self-referential "
510                         "dependency in mod->deps.\n");
511                 goto err3;
512             }
513 
514             /* Scan the current modules for this dependency */
515             for (o = module_list; o != &kernel_module && o != d; o = o->next)
516                 ;
517 
518             if (o != d) {
519                 printk(KERN_ERR "init_module: found dependency that is "
520                     "(no longer?) a module.\n");
521                 goto err3;
522             }
523         }
524 
525         /* Update module references.  */
526         for (i = 0, dep = mod->deps; i < mod->ndeps; ++i, ++dep) {
527             struct module *d = dep->dep;
528 
529             dep->ref = mod;
530             dep->next_ref = d->refs;
531             d->refs = dep;
532             /* Being referenced by a dependent module counts as a
533                use as far as kmod is concerned.  */
534             d->flags |= MOD_USED_ONCE;
535         }
536 
537         /* Free our temporary memory.  */
538         put_mod_name(n_name);
539         put_mod_name(name);
```

由于 module 结构本身已经复制过来，现在只要复制除此以外的那一部分就够了。对于 i386 处理器，module_arch_init 是空操作，总是返回 0，所以没有作用。对有些处理器，复制过来的内容有一部

分可能还在高速缓存中而没有真正写入到内存中，所以要通过 flush_icache_range( )将这些内容“冲刷”到内存中去，但是对 i386 处理器而言，这并不是个问题，所以这个函数也是空操作。

前面讲过，模块之间可能有依赖关系，正在安装中的模块可能要引用其他模块中的符号。虽然在用户空间已经完成了对这些符号的连接，但现在必须验证这些模块还在内核中未被拆除。所以，这里通过一个 for 循环检验模块的 deps 数组(见 504 行)。对于数组中的每一个元素，即 module_ref 结构，一方面是检查其 dep 指针（这些指针是在用户空间的连接阶段设置好了的）并不是指向目标模块自身；另一方面是扫描 module_list，如果 dep 指针所指的 module 结构已经不在 module_list 中(见 515 行的 for 循环)，那么这次系统调用就失败了，对目标模块的安装也就失败了。在这种情况下，应用程序（如 insmod）有责任通过系统调用 delete_module( )将已经创建的 module 结构从 module_list 中删除。

通过了第一趟扫描以后，还要再来一次扫描(见 526 行)，这一次要将每个 module_ref 结构链入到所依赖模块（指针 d 指向这个模块的 module 结构）的 refs 队列中，并将结构中的 ref 指针指向正在安装中的 module 结构。这样，每个 module_ref 结构既存在于其所属模块（即正在安装中的模块）的 deps[ ]数组中，又出现于该模块所依赖的某个模块的 refs 队列中，成为连接两个模块的纽带。从一个模块的 deps[ ]数组可以找到这个模块所依赖的所有模块，而沿着它的 refs 链则可以找到所有依赖于它的模块，从而形成了模块在内核中的“关系网”。

至此，模块的安装已基本完成，为模块安装而分配的临时缓冲区 n_name 和 name 都可以释放了。剩下的大事情还有一件，那就是启动执行模块的 init_module( )函数。

[sys_init_module( )]

```
541         /* Initialize the module.  */
542         mod->flags |= MOD_INITIALIZING;
543         atomic_set(&mod->uc.usecount,1);
544         if (mod->init && (error = mod->init( )) != 0) {
545             atomic_set(&mod->uc.usecount,0);
546             mod->flags &= ~MOD_INITIALIZING;
547             if (error > 0)  /* Buggy module */
548                 error = -EBUSY;
549             goto err0;
550         }
551         atomic_dec(&mod->uc.usecount);
552 
553         /* And set it running.  */
554         mod->flags = (mod->flags | MOD_RUNNING) & ~MOD_INITIALIZING;
555         error = 0;
556         goto err0;
557 
558     err3:
559         put_mod_name(n_name);
560     err2:
561         *mod = mod_tmp;
562         strcpy((char *)mod->name, name_tmp);/* We know there is room for this*/
563     err1:
564         put_mod_name(name);
```

```
565   err0:
566       unlock_kernel( );
567       kfree(name_tmp);
568       return error;
569   }
```

不提供 init_module( )函数，也就是让 module 结构中的函数指针 init 为 NULL 也是允许的，但是那样就使模块的存在失去了意义（除占据了一些内核空间外），因为到这里为止内核还没有任何途径访问模块中的变量或调用模块中的任何函数。所以，在正常情况下都会提供模块的 init_module( )函数。如果这个函数正常完成了它的操作就应该返回 0，否则模块的安装就失败了。

读者在后面将会看到一个典型 init_module( )函数的代码。

最后一个系统调用是 delete_module( )，用来拆卸已安装的模块，其内核中的实现为 sys_delete_module( )。我们仍旧分段来读（kernel/module.c）：

```
586   asmlinkage long
587   sys_delete_module(const char *name_user)
588   {
589       struct module *mod, *next;
590       char *name;
591       long error;
592       int something_changed;
593
594       if (!capable(CAP_SYS_MODULE))
595           return -EPERM;
596
597       lock_kernel( );
598       if (name_user) {
599           if ((error = get_mod_name(name_user, &name)) < 0)
600               goto out;
601           if (error == 0) {
602               error = -EINVAL;
603               put_mod_name(name);
604               goto out;
605           }
606           error = -ENOENT;
607           if ((mod = find_module(name)) == NULL) {
608               put_mod_name(name);
609               goto out;
610           }
611           put_mod_name(name);
612           error = -EBUSY;
613           if (mod->refs != NULL)
614               goto out;
615
616           spin_lock(&unload_lock);
```

```
617             if (!__MOD_IN_USE(mod)) {
618                 mod->flags |= MOD_DELETED;
619                 spin_unlock(&unload_lock);
620                 free_module(mod, 0);
621                 error = 0;
622             } else {
623                 spin_unlock(&unload_lock);
624             }
625             goto out;
626         }
627
628         /* Do automatic reaping */
629     restart:
630         something_changed = 0;
631         for (mod = module_list; mod != &kernel_module; mod = next) {
632             next = mod->next;
633             spin_lock(&unload_lock);
634             if (mod->refs == NULL
635                 && (mod->flags & MOD_AUTOCLEAN)
636                 && (mod->flags & MOD_RUNNING)
637                 && !(mod->flags & MOD_DELETED)
638                 && (mod->flags & MOD_USED_ONCE)
639                 && !__MOD_IN_USE(mod)) {
640                 if ((mod->flags & MOD_VISITED)
641                     && !(mod->flags & MOD_JUST_FREED)) {
642                     spin_unlock(&unload_lock);
643                     mod->flags &= ~MOD_VISITED;
644                 } else {
645                     mod->flags |= MOD_DELETED;
646                     spin_unlock(&unload_lock);
647                     free_module(mod, 1);
648                     something_changed = 1;
649                 }
650             } else {
651                 spin_unlock(&unload_lock);
652             }
653         }
654         if (something_changed)
655             goto restart;
656         for (mod = module_list; mod != &kernel_module; mod = mod->next)
657             mod->flags &= ~MOD_JUST_FREED;
658         error = 0;
659     out:
660         unlock_kernel( );
661         return error;
662     }
```

与前几个系统调用一样，只有特权用户方可允许拆卸模块。拆卸模块有两种方式：参数 name_user

为非 0，表示拆除一个特定的模块；为 0 则表示“清理仓库”，即拆除所有可以拆除的模块。我们先看有具体目标的拆除。从用户空间复制模块名之后，就通过 find_module( )从 module_list 中找到目标模块的 module 结构。模块的拆卸也是有条件的，如果内核中还有其他模块依赖于目标模块，即引用目标模块中的符号，那就不能把目标模块拆除。怎么知道是否有模块依赖于目标模块呢？看目标模块的 refs 指针是否为非 0 就可以了。即使已经没有模块依赖于目标模块，也还不能够说明就可以立即拆除这个模块，这里还有一个条件，即__MOD_IN_USE(mod)的值为 0，也就是说目标模块不在使用中。宏操作__MOD_IN_USE( )定义于 include/linux/module.h：

```
147     #define __MOD_IN_USE(mod)                                   \
148         (mod_member_present((mod), can_unload) && (mod)->can_unload \
149          ? (mod)->can_unload( ) : atomic_read(&(mod)->uc.usecount))
```

从上列宏定义可以看出，如果 module 结构中包括了函数指针 can_unload，并且这个函数指针非 0，就调用这个函数，并根据其返回值来决定是否可以拆除；否则就看 module 结构中的使用计数 uc.usecount。在前面 sys_init_module( )的代码中，我们看到计数器 uc.usecount 在调用 init_module( )之前设置成 1，而在调用之后，则将其递降成 0（见 543 行和 551 行）。这个计数器为非 0 表示正在对模块进行某种操作，所以此时若要求拆卸便失败而返回出错代码 EBUSY。反之，如果一切顺利，就调用 module.c 中的函数 free_module( )实施拆除：

[sys_delete_module( ) > free_module( )]

```
1012    /*
1013     * Free the given module.
1014     */
1015
1016    void
1017    free_module(struct module *mod, int tag_freed)
1018    {
1019        struct module_ref *dep;
1020        unsigned i;
1021
1022        /* Let the module clean up.  */
1023
1024        if (mod->flags & MOD_RUNNING)
1025        {
1026            if(mod->cleanup)
1027                mod->cleanup( );
1028            mod->flags &= ~MOD_RUNNING;
1029        }
1030
1031        /* Remove the module from the dependency lists.  */
1032
1033        for (i = 0, dep = mod->deps; i < mod->ndeps; ++i, ++dep) {
1034            struct module_ref **pp;
1035            for (pp = &dep->dep->refs; *pp != dep; pp = &(*pp)->next_ref)
```

```
1036                continue;
1037            *pp = dep->next_ref;
1038            if (tag_freed && dep->dep->refs == NULL)
1039                dep->dep->flags |= MOD_JUST_FREED;
1040        }
1041
1042        /* And from the main module list.  */
1043
1044        if (mod == module_list) {
1045            module_list = mod->next;
1046        } else {
1047            struct module *p;
1048            for (p = module_list; p->next != mod; p = p->next)
1049                continue;
1050            p->next = mod->next;
1051        }
1052
1053        /* And free the memory.  */
1054
1055        module_unmap(mod);
1056    }
```

与安装模块时调用 init_module( )相对应，拆除模块时首先要调用目标模块的 cleanup_module( )函数（1027 行）。通常，在这个函数中要将模块在系统中的“登记”撤销，使系统不能再看到这个模块的存在。读者以后会看到典型的 cleanup_module( )函数代码。调用了这个函数以后，就失去了进入这个模块的途径，从而在概念上这个模块已经不在运行中了，所以把模块的 MOD_RUNNING 标志位清 0。

拆除一个模块以后，它所依赖的所有其他模块就都少了一个“用户”。如前所述，每个模块都有一个 refs 队列，它的所有“用户”模块都通过一个 module_ref 数据结构链入到这个队列中。所以，现在要把目标模块从它所依赖的所有模块的 refs 队列中脱链。这里的 for 循环（1033 行），是检查目标模块的 deps 数组，依次处理数组中的每个 module_ref 结构，结构中的指针 dep 指向所依赖的模块（它的 module 结构）。所以，dep->dep 指向其中某个模块的 module 结构，而 dep->dep->refs 进而指向其 refs 队列。第二层 for 循环（1035 行）顺着这个队列找到指向上述 module_ref 结构的指针，使 pp 指向这个指针。由于 module_ref 结构中的指针 next_ref 指向该队列中的下一个 module_ref 结构，1037 行就使属于目标模块的 module_ref 结构从队列中脱链。这里之所以需要第二个 for 循环是因为 refs 队列是单链队列，只有顺着这个队列才能找到需要脱链的数据结构。从队列中脱链的 module_ref 结构并不需要(也不应该)单独释放，因为整个 deps 数组都是随模块的 module 结构（以及模块映象的空间）一起作为一个整体分配的。所以，最后将目标模块的 module 结构也从 module_list 队列中脱链以后，就通过 module_unmap( )将模块所占用的所有内存资源作为一个整体释放（1055 行）。

一个模块的拆除有可能使它所依赖的其他模块得到“自由”。如果将一个 module_ref 结构从一个模块的 refs 队列脱链以后使这个队列变成了空队列，那么这个模块就再没有其他模块依赖于它了。在这种情况下，如果参数 tag_freed 为非 0，就要设置这个模块的 MOD_JUST_FREED 标志位，表示这个模块刚刚得到自由。不过，在 sys_delete_module( )中有目标地拆除个别模块而调用 free_module( )时这个参数为 0（见 620 行）。

回到 sys_delete_module( )的代码中，如果调用时的参数 name_user 为 NULL，就表示要拆除内核中所有可以拆除的模块。另一方面，在有目标地拆除了一个模块之后，由于可能有模块得到了自由，也要进一步拆除所有可以拆除的模块。所谓“可以拆除”，是指同时满足以下条件：

(1) 不再有任何模块依赖于它（634 行）。

(2) 安装时带有 MOD_AUTOCLEAN 标志位，允许自动拆除（635 行）。

(3) 处于运行状态，即已经安装完毕但尚未拆除（636 行）。

(4) 尚未开始拆除（637 行），注意在上面的 618 行已经设置了模块的 MOD_DELETED 标志，然后在解了锁以后才调用 free_module( )，而 MOD_RUNNING 标志位则是在 free_module( )里面在调用了模块的 cleanup_module( )函数以后才清 0 的。所以 MOD_DELETED 标志表示拆除的过程已经启动。

(5) 模块安装以后已经受到过引用（638 行）。这种引用既可能来自其他模块，也可能来自内核。例如，在安装一个文件系统（指设备）时，如果所涉及的文件系统（指格式）是以模块方式实现的，则就要对此模块的 module 结构调用一个函数 try_inc_mod_count( )，递增其使用计数并设置其 MOD_USED_ONCE 标志位。

(6) 模块已不在使用中（639 行）。

代码中通过一个 for 循环扫描 module_list 队列，寻找同时符合所有这些条件的模块。这样的扫描可能要反复多次，因为每拆除一个模块就有可能解放出其他一些模块，而使其中的某些模块也满足了自动拆除的条件。所以，每拆除一个模块以后就把 something_changed 置成 1，使得在 for 循环结束以后再转回 restart 标号处（629 行）开始对 module_list 新一轮的扫描。

特权用户可以执行实用程序/sbin/insmod 和/sbin/rmmod，通过内核提供的上述 4 个系统调用完成具体模块的安装和拆卸。由于/sbin/insmod 实际上还是个相当复杂的目标代码连接过程，用户开发的程序中一般不应该、也不必要自行直接调用 create_module( )、init_module( )等系统调用。

在由用户通过/sbin/insmod 安装模块的过程中，内核处于一种被动的地位。但是，在很多情况下内核需要主动地启动某个模块的安装，而不能只是消极地等待。例如，读者在第 4 章关于系统调用 exec( )的代码中曾经看到，当内核打开一个特殊格式的二进制可执行程序，却发现内核中并没有支持这种格式的目标代码装入程序时，就会试图先装入支持这种格式的可安装模块。具体的代码在 exec.c 中的 search_binary_handler( )中，读者不妨回过去看一下。类似的情况还有很多，例如当内核从网络中接收到一个特殊的 packet 或报文，而支持相应规程的模块尚未安装。又如当内核检测到某种硬件，而支持这种硬件的模块尚未安装，等等。

在这样的情况下，内核都通过一个函数 reguest_module( )主动地启动模块的安装。所以，许多模块的安装实际上都是在用户不知不觉中由内核自行启动/sbin/insmod 安装的。让我们来看看这个自动安装的过程。函数 reguest_module( )的代码在 kernel/kmod.c 中：

```
159  /**
160   * request_module - try to load a kernel module
161   * @module_name: Name of module
162   *
163   * Load a module using the user mode module loader. The function returns
164   * zero on success or a negative errno code on failure. Note that a
```

```
165      *  successful module load does not mean the module did not then unload
166      *  and exit on an error of its own. Callers must check that the service
167      *  they requested is now available not blindly invoke it.
168      *
169      *  If module auto-loading support is disabled then this function
170      *  becomes a no-operation.
171      */
172
173     int request_module(const char * module_name)
174     {
175         pid_t pid;
176         int waitpid_result;
177         sigset_t tmpsig;
178         int i;
179         static atomic_t kmod_concurrent = ATOMIC_INIT(0);
180     #define MAX_KMOD_CONCURRENT 50  /* Completely arbitrary value - KAO */
181         static int kmod_loop_msg;
182
183         /* Don't allow request_module( ) before the root fs is mounted!  */
184         if ( ! current->fs->root ) {
185             printk(KERN_ERR "request_module[%s]: Root fs not mounted\n",
186                 module_name);
187             return -EPERM;
188         }
189
190         /* If modprobe needs a service that is in a module, we get a recursive
191          * loop.  Limit the number of running kmod threads to max_threads/2 or
192          * MAX_KMOD_CONCURRENT, whichever is the smaller.  A cleaner method
193          * would be to run the parents of this process, counting how many times
194          * kmod was invoked.  That would mean accessing the internals of the
195          * process tables to get the command line, proc_pid_cmdline is static
196          * and it is not worth changing the proc code just to handle this case.
197          * KAO.
198          */
199         i = max_threads/2;
200         if (i > MAX_KMOD_CONCURRENT)
201             i = MAX_KMOD_CONCURRENT;
202         atomic_inc(&kmod_concurrent);
203         if (atomic_read(&kmod_concurrent) > i) {
204             if (kmod_loop_msg++ < 5)
205                 printk(KERN_ERR
206                        "kmod: runaway modprobe loop assumed and stopped\n");
207             atomic_dec(&kmod_concurrent);
208             return -ENOMEM;
209         }
210
211         pid = kernel_thread(exec_modprobe, (void*) module_name, 0);
212         if (pid < 0) {
```

```
213             printk(KERN_ERR "request_module[%s]: fork failed, errno %d\n",
                                                    module_name, -pid);
214             atomic dec(&kmod_concurrent);
215             return pid;
216         }
217
218         /* Block everything but SIGKILL/SIGSTOP */
219         spin_lock_irq(&current->sigmask_lock);
220         tmpsig = current->blocked;
221         siginitsetinv(&current->blocked, sigmask(SIGKILL)|sigmask(SIGSTOP));
222         recalc_sigpending(current);
223         spin unlock_irq(&current->sigmask_lock);
224
225         waitpid_result = waitpid(pid, NULL, __WCLONE);
226         atomic_dec(&kmod_concurrent);
227
228         /* Allow signals again.. */
229         spin_lock_irq(&current->sigmask_lock);
230         current->blocked = tmpsig;
231         recalc_sigpending(current);
232         spin unlock_irq(&current->sigmask lock);
233
234         if (waitpid result != pid) {
235             printk(KERN_ERR
                   "request_module[%s]: waitpid(%d,...) failed, errno %d\n",
236                    module_name, pid, -waitpid_result);
237         }
238         return 0;
239     }
```

首先检查文件系统的根设备是否已经安装。由内核启动的模块安装也要靠/sbin/insmod 来完成，因此它在根设备尚未安装前显然是无法进行的。注意，reguest_module( )是在当前进程的上下文中（而不是作为中断服务程序）执行的，这样才能使当前进程进入睡眠而等待模块安装的完成。对reguest_module( )的调用有可能会嵌套，因为在安装的过程中可能会发现必须先安装另一个模块。因此，就要对嵌套深度加以限制，程序中设置了一个静态变量 kmod_concurrent，作为嵌套深度的计数器，并且还规定了嵌套深度的上限 MAX_KMOD_CONCURRENT。不过，对嵌套深度的控制还要考虑到系统中对进程数量的限制，即 max_threads，因为在安装的过程中要创建临时的进程。

通过了这些检查以后，就调用 kernel_thread( )创建一个内核线程 exec_modprobe( )。对于kernel_thread( )本身，读者已经在第 4 章中看到过了，所以我们继续往下先把 reguest_module( )代码的剩余部分读完，然后再回过来看 exec_modprobe( )的代码。

创建内核线程成功以后，先把当前进程的信号机制中除 SIGKILL 和 SIGSTOP 以外所有的信号都屏蔽掉，免得在等待模块安装的过程中受到干扰，然后就通过 waitpid( )使当前进程进入睡眠，静候佳音，等待创建出来的内核线程在完成模块安装以后退出舞台。当前进程被唤醒而从 waitpid( )返回时，内核线程 exec_modprobe( )的运行已经结束，恢复了当前进程原有的信号设置以后，根据返回值可以判

断 exec_modprobe( )的操作是否成功。如果失败，就通过 printk( )在系统的“运行日志”/var/log/messages 中登录一条出错信息。

好，现在我们来看 exec_modprobe( )的代码。它还是在 kernel/kmod.c 中：

```
139    /*
140        modprobe_path is set via /proc/sys.
141    */
142    char modprobe_path[256] = "/sbin/modprobe";
143
144    static int exec_modprobe(void * module_name)
145    {
146        static char * envp[ ] = { "HOME=/", "TERM=linux",
                                "PATH=/sbin:/usr/sbin:/bin:/usr/bin", NULL };
147        char *argv[ ] = { modprobe_path, "-s", "-k", "--", (char*)module_name, NULL };
148        int ret;
149
150        ret = exec_usermodehelper(modprobe_path, argv, envp);
151        if (ret) {
152            printk(KERN_ERR
153                   "kmod: failed to exec %s -s -k %s, errno = %d\n",
154                   modprobe_path, (char*) module_name, errno);
155        }
156        return ret;
157    }
```

这里的 modprobe_path 就是路径名/sbin/modprobe，所以，如果 module_name 为 mymodule 的话，这里的 argv[ ]就相应于命令：

```
/sbin/modprobe   -s  -k  mymodule
```

选择项-s 表示在安装过程中产生的信息应写入系统的运行日志，而不要在控制终端上显示；-k 表示安装时将模块的 MOD_AUTOCLEAN 标志位设成 1。

函数 exec_usermodehelper( )的代码也在同一文件(kmod.c)中：

[exec_modprobe( ) > exec_usermodehelper( )]

```
86    int exec_usermodehelper(char *program_path, char *argv[ ], char *envp[ ])
87    {
88        int i;
89        struct task_struct *curtask = current;
90
91        curtask->session = 1;
92        curtask->pgrp = 1;
93
94        use_init_fs_context( );
95
96        /* Prevent parent user process from sending signals to child.
```

```
97           Otherwise, if the modprobe program does not exist, it might
98           be possible to get a user defined signal handler to execute
99           as the super user right after the execve fails if you time
100          the signal just right.
101       */
102       spin_lock_irq(&curtask->sigmask_lock);
103       sigemptyset(&curtask->blocked);
104       flush_signals(curtask);
105       flush_signal_handlers(curtask);
106       recalc_sigpending(curtask);
107       spin_unlock_irq(&curtask->sigmask_lock);
108
109       for (i = 0; i < curtask->files->max_fds; i++ ) {
110           if (curtask->files->fd[i]) close(i);
111       }
112
113       /* Drop the "current user" thing */
114       {
115           struct user_struct *user = curtask->user;
116           curtask->user = INIT_USER;
117           atomic_inc(&INIT_USER->__count);
118           atomic_inc(&INIT_USER->processes);
119           atomic_dec(&user->processes);
120           free_uid(user);
121       }
122
123       /* Give kmod all effective privileges.. */
124       curtask->euid = curtask->fsuid = 0;
125       curtask->egid = curtask->fsgid = 0;
126       cap_set_full(curtask->cap_effective);
127
128       /* Allow execve args to be in kernel space. */
129       set_fs(KERNEL_DS);
130
131       /* Go, go, go... */
132       if (execve(program_path, argv, envp) < 0)
133           return -errno;
134       return 0;
135   }
```

这个函数是在内核线程 exec_modprobe( )的上下文中运行的，所以这里的 current 指向这个内核线程的 task_struct 结构，而与创建这个线程时的 current 不同，那时候的 current 指向当时的当前进程，即 exec_modprobe( )的父进程。内核线程 exec_modprobe( )从其父进程那里继承了绝大部分资源和特性，包括它的 fs_struct 的内容和所有的已打开文件，以及它的进程号、组号，还有所有的特权。但是，这些从父进程继承下来的资源和特性未必满足模块安装的要求。首先是父进程的根目录，在“文件系统”一章中讲过，一个进程可以设置自身的根目录，所以父进程的根目录有可能并不是真正的整个文件系

统的根目录。如果那样的话，在父进程的根目录中也许根本就没有 sbin 这么个子目录，更找不到 modprobe 了。所以，首先得要确保这个内核线程的 fs_srtuct 中的指针 root 确实指向文件系统的总根，结构中的其他一些内容也要与其相一致。怎么来保证这一点呢？系统中只有一个进程，即 init_task 的根目录是肯定不会改变的，所以这里通过 kmod.c 中的一个函数 use_init_fs_context( )从 init_task 结构中直接继承其根目录等资源。

[exec_modprobe( ) > exec_usermodehelper( ) > use_init_fs_context( )]

```
32    static inline void
33    use_init_fs_context(void)
34    {
35        struct fs struct *our_fs, *init_fs;
36        struct dentry *root, *pwd;
37        struct vfsmount *rootmnt, *pwdmnt;
38
39        /*
40         * Make modprobe's fs context be a copy of init's.
41         *
42         * We cannot use the user's fs context, because it
43         * may have a different root than init.
44         * Since init was created with CLONE_FS, we can grab
45         * its fs context from "init_task".
46         *
47         * The fs context has to be a copy. If it is shared
48         * with init, then any chdir( ) call in modprobe will
49         * also affect init and the other threads sharing
50         * init task's fs context.
51         *
52         * We created the exec_modprobe thread without CLONE_FS,
53         * so we can update the fields in our fs context freely.
54         */
55
56        init_fs = init task.fs;
57        read_lock(&init_fs->lock);
58        rootmnt = mntget(init_fs->rootmnt);
59        root = dget(init fs->root);
60        pwdmnt = mntget(init_fs->pwdmnt);
61        pwd = dget(init_fs->pwd);
62        read_unlock(&init fs->lock);
63
64        /* FIXME - unsafe ->fs access */
65        our fs = current->fs;
66        our fs->umask = init fs->umask;
67        set fs_root(our_fs, rootmnt, root);
68        set fs_pwd(our_fs, pwdmnt, pwd);
69        write lock(&our_fs->lock);
70        if (our fs->altroot) {
```

```
71             struct vfsmount *mnt = our_fs->altrootmnt;
72             struct dentry *dentry = our_fs->altroot;
73             our_fs->altrootmnt = NULL;
74             our_fs->altroot = NULL;
75             write_unlock(&our_fs->lock);
76             dput(dentry);
77             mntput(mnt);
78         } else
79             write_unlock(&our_fs->lock);
80         dput(root);
81         mntput(rootmnt);
82         dput(pwd);
83         mntput(pwdmnt);
84     }
```

对这一段代码就不需要多说了（读者若感到困难就应该复习一下第5章中的有关内容）。解决了根目录的问题以后，还要把从父进程继承的待处理信号全都“冲刷”掉，即全部丢弃，并且把父进程所设置的信号处理也全部恢复成由系统默认的处理方式。然后，继承下来的全部已打开文件也要关闭，连打开文件号0、1、2所代表的三个标准输入/输出和出错输出文件也都关闭，这样就与父进程的控制终端脱离了关系。不但如此，连线程的uid、euid、fsuid也要全部换成0，使其变成特权（超级）用户进程，还要通过cap_set_full( )赋予其全部特权（第126行）。读者也许感到迷惑，怎么这个进程有这么大的能耐，竟然可以为所欲为，不受限制？原因就在于这是个内核线程，而不是进程。

第129行的set_fs( )是个宏操作，定义于include/asm_i386/uaccess.h:

```
30    #define set_fs(x)    (current->addr_limit = (x))
```

也就是说，把从父进程继承的地址上限也换成KERNEL_DS，即0xffffffff(4GB)，因为父进程的地址上限很可能是USER_DS，即0xbfffffff(3GB)。

经过这么一番脱胎换骨的改造，内核线程 exec_modprobe( )就成了一个彻底的特权用户进程（线程），而与原先的父进程没有多大联系了。套用一句时下流行的话，这叫“借壳上市”。但是，两个进程（线程）间的父子关系还在，当exec_modproc( ) 死亡退出运行时还是会唤醒其父进程。

最后，就是通过execve( )执行/sbin/modprobe了，有关ececve( )的详情，包括运行结束时的处理，可参阅第4章。

在阅读模块的init_module( )和cleanup_module( )两个函数之前，还要讲一下内核中符号的“移出”问题，也就是内核中的哪一些符号可以被可安装模块引用。内核中的每一个符号都必须通过宏定义 EXPORT_SYMBOL 明确规定准予移出，才能由/sbin/insmod 通过query_module( )系统调用获得这些符号的地址，否则就都是不准移出的。下面是取自kernel/ksyms.c中的几个例子：

```
142    EXPORT_SYMBOL(path_init);
143    EXPORT_SYMBOL(path_walk);
144    EXPORT_SYMBOL(path_release);
```

```
145    EXPORT_SYMBOL(__user_walk);
146    EXPORT_SYMBOL(lookup_one);
147    EXPORT_SYMBOL(lookup_hash);
148    EXPORT_SYMBOL(sys_close);
```

宏定义 EXPORT_SYMBOL 将给定符号的符号名和它的值，即该符号在连接后的内核映象中的地址、组装在一个 module_symbol 数据结构中，并指示连接程序（ld）在连接内核映象时将这个结构放在一个称为“__ksymtab”的区段中。这样，连接以后所有这样的数据结构就都在__ksymtab 区段中，而这个区段就成了内核对外公开的符号表。而“对内”的符号表则由连接程序自行生成，并仅供连接程序 ld 自己使用。数据结构类型 module_symbol 是在 include/linux/module.h 中定义的：

```
37    struct module_symbol
38    {
39        unsigned long value;
40        const char *name;
41    };
```

这里的 name 只是一个字符指针，真正的字符则存放在另一个区段“.kstrtab”中。这样做的好处在于使符号表的结构变得规则，因为不管符号名字符串有多长，指针的大小总是固定的。

这里还要指出，内核对可安装模块的支持是可选的。如果在编译内核代码之前的系统配置（config）阶段选择了支持可安装模块，就定义了编译提示 CONFIG_MODULES，使支持可安装模块的代码受到编译，而 EXPORT_SYMBOL 也只有在这种情况下才有定义。

与 EXPORT_SYMBOL 有关的定义都在 include/linux/module.h 中：

```
151    /* Indirect stringification.  */
152
153    #define __MODULE_STRING_1(x)    #x
154    #define __MODULE_STRING(x)  __MODULE_STRING_1(x)

325    #define __EXPORT_SYMBOL(sym, str)            \
326    const char __kstrtab_##sym[ ]                 \
327    __attribute__((section(".kstrtab"))) = str;     \
328    const struct module_symbol __ksymtab_##sym      \
329    __attribute__((section("__ksymtab"))) =            \
330    { (unsigned long)&sym, __kstrtab_##sym }
331
332    #if defined(MODVERSIONS) || !defined(CONFIG_MODVERSIONS)
333    #define EXPORT_SYMBOL(var)  __EXPORT_SYMBOL(var, __MODULE_STRING(var))
334    #else
335    #define EXPORT_SYMBOL(var) \
           __EXPORT_SYMBOL(var, __MODULE_STRING(__VERSIONED_SYMBOL(var)))
336    #endif
```

第 332～336 行是条件编译，即在不同条件下对 EXPORT_SYMBOL 有不同的定义。为什么要有不同呢？这是为了保证内核与可安装模块在版本上的严格一致。要保证二者的版本严格一致，最好的办

法就是将版本信息编码进变量名中，例如将版本号作为变量名的后缀。这样，如果变量相同而版本号不一致，/sbin/insmod 就会认为是两个不同的符号而不予连接。但是，另一方面，这在一定程度上也带来了不便。因为每当有版本变动时就要重新编译（或从网上下载）许多可安装模块，而有些版本变动实际上只是很小的变动，并不影响运行。因此，内核把是否将版本信息编码进符号名作为一个可选项提供。如果需要那样做的话，就可以在编译内核代码前的系统配置阶段选择这个可选项，而使条件编译提示 CONFIG_MODVERSIONS 成为有定义。有时候，虽然从总体上选择了使用版本信息，但对于某些源代码和模块中的符号却并不需要搞得这么复杂，这时就可以在编译这些源代码时在命令行中加上“-D MODVERSIONS”可选项，或在源代码文件中加上“#define MODVERSIONS”，使该文件中定义的符号摆脱可选项 CONFIG_MODVERSIONS 的控制。

我们先看不带版本信息的符号表项是如何建立的。此时的宏操作 EXPORT_SYMBOL 定义于第 333 行。以符号 path_init 为例，此时 EXPORT_SYMBOL(path_init) 的定义就成了 __EXPORT_SYBOL(path_init，“path_init”)。而第 325 行又进而将其定义为两个语句，第一个语句（326～327 行）定义了一个名为__kstrtab_path_init 的字符串，将字符串的内容初始化为“path_init”，并将其置于内核映象中的.kstrtab 区段。第二个语句（328～330 行）则定义了一个名为 __ksymtab_path_init 的 module_symbol 数据结构，将其初始化成{&path_init，__kstrtab_path_init}，并将其置于内核映象中的_ksymtab 区段。这样，这个数据结构中的字段 value 的值为 path_init 在内存映象中的地址，而指针 name 则指向字符串“path_init”。顺便提一下，这里定义的两项数据都是 const，初始化以后便不容改变。

采用版本信息时的操作就要复杂一些了。不同之处在于__EXPORT_SYMBOL 的第二个参数先要经过另一个宏操作__VERSIONED_SYMBOL 的变换，这个宏操作定义于 include/linux/modsetver.h:

```
3    #define __SYMBOL_VERSION(x)          __ver_ ## x
4    #define __VERSIONED_SYMBOL2(x, v)    x ## _R ## v
5    #define __VERSIONED_SYMBOL1(x, v)    __VERSIONED_SYMBOL2(x, v)
6    #define __VERSIONED_SYMBOL(x)        \
                        __VERSIONED_SYMBOL1(x, __SYMBOL_VERSION(x))
```

仍以 path_init 为例，宏操作__SYMBOL_VERSION(path_init)首先将其变换成__ver_path_init；然后__VERSIONED1 又将 path_init、_R 以及__ver_path_init 三部分连在一起。可是，经过这样的处理，最后形成的字符串中并没有版本信息的编码啊！奥妙就在于后缀__ver_path_init 并不是最终用来与前面两部分相连的字符串，它也还只是一个中间产物。在某个源代码文件中还会有类似于“#define __ver_path_init smp_1234abcd”这样的宏定义。这样，把三部分连在一起以后就形成了类似于“path_init_Rsmp_1234abcd”这样的字符串（这里 smp 表示内核支持 SMP 多处理器结构），最后字符串__kstrtab_path_init 的内容就成为“path_init_Rsmp_1234abcd”，而数据结构__ksymtab_path_init 中的内容则仍为{&path_init，__kstrtap_path_init}。显然，_Rsmp_1234abcd 即为包含着版本信息的符号名后缀。那么，这个后缀是怎么来的呢？这是由一个工具/sbin/genksyms 根据版本号和具体符号的类型编码（CRC）产生的。熟悉 C++的读者一定会联想起 C++中对符号（变量名或函数名）的编码（称为 mangling）。读者可以在内核的 Makefile 和 Rules.make 中找到对这个工具的应用，还可以用命令“man genksyms”看一下这个工具的说明。由工具产生的输出写入到一些.ver 文件中，这些.ver 文件则又被 include 到 include/linux/modversions.h 中。这个.h 文件是在通过 make 生成内核映象的过程中生成的，看一下它的

具体内容，我们就可以发现大量的类似于“#include　<linux/ksyms.ver>”这样的“语句”。

只要内核映象中的__ksymtab 区段非空，内核就有对外公开的符号，可以通过系统调用 query_module( )查询。此外，在/proc 目录中有个特殊文件 ksyms，用来提供内核中的“移出”符号清单（包括已安装模块的符号）。读者不妨用命令“more　/proc/ksyms”看一下你的内核符号是否包含版本信息编码。

现在，我们可以来看几个典型模块的 init_module( )和 cleanup_module( )函数了。

我们要看的第一个模块是一个声卡驱动程序，名为“sparcaudio”。我们的目的并不在于了解具体的驱动程序代码，而主要在于了解它在 init_module( )中向系统登记、从而建立起从内核中文件系统的 vfs 层进入其具体驱动程序的途径这么一个过程，以及在 cleanup_module( )中将这个途径撤销的过程。另一个使我们对它感兴趣的原因是，它还并不是真正直接驱动声卡硬件的模块，而只是 vfs 层与物理层之间的一个中间层次。它一方面向系统登记，建立起从 vfs 层进入它的途径；另一方面又准备接受来自更低层模块的登记，从而建立起进入物理层的途径。像这样由不同的模块来实现一种系统结构中的不同层次，从而形成一个模块“堆栈”的做法，是一种典型的程序结构和实现方式（尤其是在计算机网络环境下）。同时，它也是模块间依赖关系的一个实例。这个模块的代码在 drivers/sbus/audio/audio.c 中。路径名中的 drivers/sbus 表示有关的代码都是 SUN 公司的 Sparc 结构工作站中 sbus 总线设备的驱动程序。我们不准备深入到 sbus 的细节中去，而只是用它作为一个可安装模块的实例。

对 sparcaudio 的支持既可以编译成一个可安装模块，也可以静态地编译连接进内核的映象。内核编译之前的系统配置阶段为用户提供了选择手段。通常，对每一种设备或功能都有一个问题让用户回答，这个问题就是要或者不要、以及以何种方式支持某种设备或功能。如果回答“Y”就表示要，并且将相应的代码静态地连接在内核映象中；回答“M” 表示要，但是将代码编译成可安装模块；否则就回答“N”,表示不要。当选择将代码编译成可安装模块时，模块的入口就是 init_module( )，就好像每个可执行程序的入口都是 main( )一样。虽然每个模块都有一个 init_module( )，但是这些模块不会由 ld 连接在一起，所以不会互相冲突。但是，如果将代码编译并连接进内核映象，那就不能使用 init_modle( )这个函数名了，那样会造成符号名冲突而不能连接。所以，对 sparcaudio 不采用可安装模块方式时，就要给 init_module( )函数换个名字，这里叫 sparcaudio_init( )，并且加上了说明__init，表示这个函数是个初始化函数，同时还要在代码中加上这么一行：

```
2230    module_init(sparcaudio_init)
```

表示在系统初始化时应调用这个函数。从这里也可以看出，即使不把它编译成可安装模块，代码的设计和实现仍然是高度模块化的。

先看 sparcaudio_init( )，这个函数在 drivers/sbus/audio/audio.c 中：

```
2200    static int __init sparcaudio_init(void)
2201    {
2202        /* Register our character device driver with the VFS. */
2203        if (devfs_register_chrdev(SOUND_MAJOR, "sparcaudio", &sparcaudio_fops))
2204            return -EIO;
2205
```

```
2206        devfs_handle = devfs_mk_dir (NULL, "sound", NULL);
2207
2208    #ifdef CONFIG_SPARCAUDIO_AMD7930
2209        amd7930_init( );
2210    #endif
2211    #ifdef CONFIG_SPARCAUDIO_DBRI
2212            dbri_init( );
2213    #endif
2214    #ifdef CONFIG_SPARCAUDIO_CS4231
2215        cs4231_init( );
2216    #endif
2217    #ifdef CONFIG_SPARCAUDIO_DUMMY
2218        dummy_init( );
2219    #endif
2220
2221        return 0;
2222    }
```

前面提到过，当以不同的模块实现一个系统结构中的不同层次时，会形成一个"模块堆栈"。之所以称之为"堆栈"，一方面是因为这是一组模块，另一方面是这些模块也遵循"后进先出"的原则，位于堆栈（层次意义上的）顶部的模块其实就是内核本身，虽然它并不是一个"可安装模块"。从这个意义上说，每一个模块实际上都存在于一个模块堆栈中。我们来考察一下模块堆栈中相邻两层间的界面。首先，位于上层的模块肯定要调用下层模块中的函数。这个函数调用的界面是标准化的，那就是类似于 file_operations 一类包含着函数指针的数据结构，由下层模块通过向其上层"登记"的方式递交给上层。除此以外，上层模块就不能、也不应直接调用下层模块中的函数，或访问下层模块中的变量了。也就是说，上层模块不直接引用下层模块的符号，对下层模块没有依赖关系。可是，反过来下层模块对于上层模块就有依赖关系了。每一个模块都不能脱离它的环境而存在，而运行。这个环境就是由位于其上层的模块提供的。至少，它的直接上层应提供一个函数让它可以向上层登记，所以至少要向它移出一个符号。这样，每个模块对位于其上层的模块就有了依赖关系，而除最底层以外的模块就有必要向位于其下层的模块移出若干符号。那么，是否每个模块都直接依赖于位于其上层的所有模块呢？那倒不一定，但是一般对"顶头上司"和位于最高层的内核都有直接的依赖关系，也就是需要直接引用这些模块中的符号。由于 sparcaudio 并不是处于最底层的模块，它也要向它下层的模块移出一些符号。对于 2.1.0 以前的版本，模块要通过一个函数 register_symtab( )向内核登记它的符号表，而在从那以后的版本中，对模块符号表的处理在形式上就与内核一致了。模块 sparcandio 的符号表定义为：

```
2195    EXPORT_SYMBOL(register_sparcaudio_driver);
2196    EXPORT_SYMBOL(unregister_sparcaudio_driver);
2197    EXPORT_SYMBOL(sparcaudio_output_done);
2198    EXPORT_SYMBOL(sparcaudio_input_done);
```

这个模块向低层模块移出的符号有 4 个，都是供位于其下层的模块调用的函数（从符号表中并不能看出具体的符号是函数还是变量，这是由模块间的界面决定的），其中前两个显然是供下层模块登记和撤销登记时调用的函数。后两个则从函数名可以看出是供低层模块在完成输入/输出后"回叫"的函

数。

向上层的登记是通过 devfs_register_chrdev( )进行的，这个函数由内核提供并移出，定义于 fs/devfs/base.c：

[sparcaudio_init( ) > devfs_register_chrdev( )]

```
1935    int devfs_register_chrdev (unsigned int major, const char *name,
1936                        struct file_operations *fops)
1937    {
1938        if (boot_options & OPTION_ONLY) return 0;
1939        return register_chrdev (major, name, fops);
1940    }   /*  End Function devfs_register_chrdev  */
```

登记时使用了三个参数，第一个参数是模块所代表设备的主设备号 SOUND_MAJOR，在 include/linux/major.h 中定义为 14：

```
  38    #define SOUND_MAJOR 14
```

第二个参数为模块名，而第三个参数就是指向该种设备的 file_operations 数据结构的指针，这个数据结构也是在 drivers/sbus/audio/audio.c 中定义的：

```
1961    static struct file_operations sparcaudio_fops = {
1962        owner:      THIS_MODULE,
1963        llseek:     sparcaudio_lseek,
1964        read:       sparcaudio_read,
1965        write:      sparcaudio_write,
1966        poll:       sparcaudio select,
1967        ioctl:      sparcaudio_ioctl,
1968        open:       sparcaudio_open,
1969        release:    sparcaudio_release,
1970    };
```

登记的是 file_operations 结构，这也说明在这个模块的上面就是内核（严格说是内核的 vfs 层），而没有其他模块了。另一方面，从函数名 devfs_register_chrdev( )也可以看出，这个模块所支持的是树状设备文件系统 devfs，并且声卡设备是字符设备。不过，devfs 的字符设备登记与普通设备文件实际上相同，函数 register_chrdev( )的代码在 fs/devices.c 中：

[sparcaudio_init( ) > devfs_register_chrdev( ) > register_chrdev( )]

```
  98    int register_chrdev(unsigned int major, const char * name,
                                            struct file_operations *fops)
  99    {
 100        if (major == 0) {
 101            write_lock(&chrdevs_lock);
 102            for (major = MAX_CHRDEV-1; major > 0; major--) {
```

```
103                 if (chrdevs[major].fops == NULL) {
104                     chrdevs[major].name = name;
105                     chrdevs[major].fops = fops;
106                     write_unlock(&chrdevs_lock);
107                     return major;
108                 }
109             }
110             write_unlock(&chrdevs_lock);
111             return -EBUSY;
112         }
113         if (major >= MAX_CHRDEV)
114             return -EINVAL;
115         write_lock(&chrdevs_lock);
116         if (chrdevs[major].fops && chrdevs[major].fops != fops) {
117             write_unlock(&chrdevs_lock);
118             return -EBUSY;
119         }
120         chrdevs[major].name = name;
121         chrdevs[major].fops = fops;
122         write_unlock(&chrdevs_lock);
123         return 0;
124     }
```

登记字符设备（以及块设备）时，如果以 0 作为主设备号，就表示要求由内核分配一个主设备号。当然，由内核分配的主设备号只是临时的。内核中有个 device_struct 结构数组 chrdevs[ ]，这是一个以字符设备的主设备号为下标的数组，每个元素都是一个 device_struct 数据结构，有关的定义在 fs/devices.c 中：

```
33    struct device_struct {
34        const char * name;
35        struct file_operations * fops;
36    };
      ......
39    static struct device_struct chrdevs[MAX_CHRDEV];
40
```

所谓登记，就是将由模块提供的 file_operations 结构指针填入这个数组（或称字符设备表）的某个表项。登记以后，只要知道了字符设备的主设备号，就可以很快找到它的 file_operations 数据结构，进而找到该种设备的各种驱动函数。

登记了以后，位于上层的模块（在这里是内核）就可以“看见”这个模块了。但是，应用程序却还不能“看见”它，因而还不能通过系统调用来使用它。要使应用程序能“看见”这个模块或者这个模块所驱动的设备，就要在文件系统中为其创建一个代表它的节点。这个节点可以是在单层的/dev 目录中的一个节点，也可以是在多层的设备文件目录下的一个节点。在单层的/dev 目录中，每个节点都是文件节点，每个节点都代表一个具体的设备，所以要有主设备号和次设备号两项参数才能创建一个节点。这一点读者在 mknod( )的代码中已经看到过了。对于多层的 devfs，应给（由主设备号确定的）

每一种设备都创建一个目录节点，在这个目录下才是代表具体设备的文件节点。函数 devfs_mk_dir( )的作用就是创建这样的目录节点。

调用这个函数时的第一个参数 dir 是一个指针，指向代表父目录节点的 devfs_entry 数据结构，当 dir 为 NULL 时表示父目录节点为 devfs 的根节点，即“/dev”。与普通目录节点的 dentry 数据结构不同，devfs 树中目录节点的数据结构为 devfs_entry，而指向这种数据结构的指针类型则定义为 devfs_handle_t。第二个参数为待创建目录节点的节点名，这里是“sound”。第三个参数则为节点名的长度，以 0 为参数表示由 devfs_mk_dir( )计算出字符串的长度。最后一个参数为指向附加信息的指针，这里是 NULL，表示并无附加信息。由于读者已经阅读过 path_walk( )、mknod( )等函数的代码，这里我们就不讲解 devfs_mk_dir( )的代码了，有兴趣的读者可自行阅读。

至此，模块 sparcaudio 的初始化实际上已经完成了。但是，这个模块并不是最底层的模块，所以还要考虑它的（直接）下层。如果它的下层也是通过可安装模块实现的，那这里就不用做什么了；因为在安装那个模块时自会调用其 init_module( )函数。可是，如果下层驱动程序是通过静态模块实现的话，那就要在这里处理下一层的初始化了。根据具体硬件的不同，sparcaudio 的下层有三种不同的驱动程序，另外为程序调试的目的还可以再加上一层虚设的驱动程序，所以这里有 4 个条件编译的语句。我们假设下层是通过可安装模块实现的，所以就跳过了这些静态模块的初始化。至于同一文件中，在撤销模块时调用的 sparcaudio_exit( )，就留给读者自己阅读了。由于这个模块是静态连接的，代码中相应地还有一行：

```
2231    module_exit(sparcaudio_exit)
```

表示在系统退出运行时要拆除这个模块。

很自然地，我们在这里要看的第二个模块就是位于 sparcaudio 下层的模块，即“amd7930”(我们假定声卡所用的芯片为 AMD7930)。这一次，我们假定 amd7930 是个动态安装模块，所以由 sys_init_module( )在安装这个模块时调用它的 init_module( )函数，其代码在 drivers/sbus/audio/amd7930.c 中：

[sys_init_module( ) > init_module( )]

```
1677    /* Probe for the amd7930 chip and then attach the driver. */
1678    #ifdef MODULE
1679    int init_module(void)
1680    #else
1681    int __init amd7930_init(void)
1682    #endif
1683    {
1684        struct sbus_bus *sbus;
1685        struct sbus_dev *sdev;
1686        int node;
1687
1688        /* Try to find the sun4c "audio" node first. */
1689        node = prom_getchild(prom_root_node);
1690        node = prom_searchsiblings(node, "audio");
```

```
1691        if (node && amd7930_attach(&drivers[0], node, NULL, NULL) == 0)
1692            num_drivers = 1;
1693        else
1694            num drivers = 0;
1695
1696        /* Probe each SBUS for amd7930 chips. */
1697        for_all_sbusdev(sdev, sbus) {
1698            if (!strcmp(sdev->prom_name, "audio")) {
1699                /* Don't go over the max number of drivers. */
1700                if (num_drivers >= MAX_DRIVERS)
1701                    continue;
1702
1703                if (amd7930_attach(&drivers[num drivers],
1704                           sdev->prom_node, sdev->bus, sdev) == 0)
1705                    num_drivers++;
1706            }
1707        }
1708
1709        /* Only return success if we found some amd7930 chips. */
1710        return (num_drivers > 0) ? 0 : -EIO;
1711    }
```

我们的目的不在于掌握 sbus 和 AMD7930 芯片本身的细节，所以就不详细深入到有关的代码中去了。从总体上说，这段程序所做的就是：对于探测到的每个 AMD7930 芯片，即音频信道，从结构数组 drivers[ ]中分配一个数据结构，然后调用 amd7931_attach( )。如果一个信道也没有探测到，那就返回－EIO 表示模块安装失败，否则就返回 0 表示安装成功。

函数 amd7931_attach( )的代码在 amd7930.c 中：

[sys_init_module( ) > init_module( ) > amd7930_attach( )]

```
1566    /* Attach to an amd7930 chip given its PROM node. */
1567    static int amd7930_attach(struct sparcaudio_driver *drv, int node,
1568                struct sbus bus *sbus, struct sbus_dev *sdev)
1569    {
1570        struct linux_prom registers regs;
1571        struct linux_prom_irqs irq;
1572        struct resource res, *resp;
1573        struct amd7930_info *info;
1574        int err;
1575
1576        /* Allocate our private information structure. */
1577        drv->private = kmalloc(sizeof(struct amd7930_info), GFP_KERNEL);
1578        if (drv->private == NULL)
1579            return -ENOMEM;
1580
1581        /* Point at the information structure and initialize it. */
1582        drv->ops = &amd7930 ops;
```

```
1583        info = (struct amd7930 info *)drv->private;
1584        memset(info, 0, sizeof(*info));
1585        info->ints_on = 1; /* force disable below */
1586
1587        drv->dev = sdev;
1588
1589        /* Map the registers into memory. */
1590        prom getproperty(node, "reg", (char *)&regs, sizeof(regs));
1591        if (sbus && sdev) {
1592            resp = &sdev->resource[0];
1593        } else {
1594            resp = &res;
1595            res.start = regs.phys addr;
1596            res.end = res.start + regs.reg size - 1;
1597            res.flags = IORESOURCE_IO | (regs.which_io & 0xff);
1598        }
1599        info->regs_size = regs.reg_size;
1600        info->regs = sbus_ioremap(resp, 0, regs.reg_size, "amd7930");
1601        if (!info->regs) {
1602            printk(KERN_ERR "amd7930: could not remap registers\n");
1603            kfree(drv->private);
1604            return -EIO;
1605        }
1606
1607        /* Put amd7930 in idle mode (interrupts disabled) */
1608        amd7930_idle(info);
1609
1610        /* Enable extended FIFO operation on D-channel */
1611        sbus_writeb(AMR_DLC_EFCR, info->regs + CR);
1612        sbus writeb(AMR_DLC_EFCR_EXTEND_FIFO, info->regs + DR);
1613        sbus writeb(AMR DLC_DMR4, info->regs + CR);
1614        sbus_writeb(/* AMR DLC DMR4 RCV 30 | */ AMR_DLC_DMR4_XMT_14,
1615                info->regs + DR);
1616
1617        /* Attach the interrupt handler to the audio interrupt. */
1618        prom_getproperty(node, "intr", (char *)&irq, sizeof(irq));
1619        info->irq = irq.pri;
1620        request_irq(info->irq, amd7930_interrupt,
1621                SA_INTERRUPT, "amd7930", drv);
1622        enable_irq(info->irq);
1623        amd7930 enable ints(info);
1624
1625        /* Initalize the local copy of the MAP registers. */
1626        memset(&info->map, 0, sizeof(info->map));
1627        info->map.mmr1 = AM_MAP_MMR1_GX | AM MAP MMR1_GER |
1628                AM_MAP_MMR1_GR | AM_MAP_MMR1 STG;
1629            /* Start out with speaker, microphone */
1630            info->map.mmr2 |= (AM_MAP_MMR2_LS | AM_MAP_MMR2 AINB);
```

```
1631
1632        /* Set the default audio parameters. */
1633        info->rgain = 128;
1634        info->pgain = 200;
1635        info->mgain = 0;
1636        info->format_type = AUDIO_ENCODING_ULAW;
1637        info->Bb.input_format = AUDIO_ENCODING_ULAW;
1638        info->Bb.output_format = AUDIO_ENCODING_ULAW;
1639        info->Bc.input_format = AUDIO_ENCODING_ULAW;
1640        info->Bc.output_format = AUDIO_ENCODING_ULAW;
1641        amd7930_update_map(drv);
1642
1643        /* Register the amd7930 with the midlevel audio driver. */
1644        err = register_sparcaudio_driver(drv, 1);
1645        if (err < 0) {
1646            printk(KERN_ERR "amd7930: unable to register\n");
1647            disable_irq(info->irq);
1648            free_irq(info->irq, drv);
1649            sbus_iounmap(info->regs, info->regs_size);
1650            kfree(drv->private);
1651            return -EIO;
1652        }
1653
1654        /* Announce the hardware to the user. */
1655        printk(KERN_INFO "amd7930 at %lx irq %d\n",
1656               info->regs, info->irq);
1657
1658        /* Success! */
1659        return 0;
1660    }
```

同样，我们的目的不在于这段程序的细节，因为那完全取决于具体的硬件和具体驱动程序的设计。我们在这里关心的主要有两件事。其一是对函数 register_sparcaudio_driver( )的调用(1644 行)，这个函数是由上层模块 sparcaudio 提供的，模块 amd7930 通过它向上层登记一个 sparcaudio_driver 结构指针，也就是(代表着)一个 AMD7930 芯片。其二(1582 行)，是使这个数据结构中的一个指针 ops，指向在采用 AMD7930 芯片条件下 sparcaudio 设备各种操作的函数跳转结构，即 sparcaudio_operations 结构 amd7930_ops:

```
1503    /*
1504     * Device detection and initialization.
1505     */
1506
1507    static struct sparcaudio_operations amd7930_ops = {
1508        amd7930_open,
1509        amd7930_release,
1510        amd7930_ioctl,
```

```
1511        amd7930_start_output,
1512        amd7930_stop_output,
1513        amd7930_start_input,
1514        amd7930_stop_input,
1515        amd7930_sunaudio_getdev,
1516        amd7930_set_output_volume,
1517        amd7930_get_output_volume,
1518        amd7930_set_input_volume,
1519        amd7930_get_input_volume,
1520        amd7930_set_monitor_volume,
1521        amd7930_get_monitor_volume,
1522        NULL,             /* amd7930_set_output_balance */
1523        amd7930_get_output_balance,
1524        NULL,             /* amd7930_set_input_balance */
1525        amd7930_get_input_balance,
1526        amd7930_set_output_channels,
1527        amd7930_get_output_channels,
1528        amd7930_set_input_channels,
1529        amd7930_get_input_channels,
1530        amd7930_set_output_precision,
1531        amd7930_get_output_precision,
1532        amd7930_set_input_precision,
1533        amd7930_get_input_precision,
1534        amd7930_set_output_port,
1535        amd7930_get_output_port,
1536        NULL,             /* amd7930_set_input_port */
1537        amd7930_get_input_port,
1538        amd7930_set_encoding,
1539        amd7930_get_encoding,
1540        amd7930_set_encoding,
1541        amd7930_get_encoding,
1542        amd7930_set_output_rate,
1543        amd7930_get_output_rate,
1544        amd7930_set_input_rate,
1545        amd7930_get_input_rate,
1546        amd7930_sunaudio_getdev_sunos,
1547        amd7930_get_output_ports,
1548        amd7930_get_input_ports,
1549        NULL,                   /* amd7930_set_output_muted */
1550        amd7930_get_output_muted,
1551            NULL,                  /* amd7930_set_output_pause */
1552            NULL,                  /* amd7930_get_output_pause */
1553            NULL,                  /* amd7930_set_input_pause */
1554            NULL,                  /* amd7930_get_input_pause */
1555            NULL,                  /* amd7930_set_output_samples */
1556            NULL,                  /* amd7930_get_output_samples */
1557            NULL,                  /* amd7930_set_input_samples */
1558            NULL,                  /* amd7930_get_input_samples */
```

```
1559            NULL,                       /* amd7930 set_output_error */
1560            NULL,                       /* amd7930_get_output_error */
1561            NULL,                       /* amd7930_set input_error */
1562            NULL,                       /* amd7930_get_input error */
1563            amd7930_get_formats,
1564    };
```

我们在这里还是列出了 amd7930_attach( )的全部代码以及 amd7930_ops 的全部定义，使读者对物理层的驱动程序有个感性的认识。即使不深入到具体的函数中去，我们也能从 amd7930_attach( )的代码中约略地看到：它将芯片中的寄存器映射到内存空间，对硬件进行初始化，为来自 AMD7930 芯片的中断请求设置好中断服务程序（amd7930_interrupt( )），并打开中断，然后再向其上层登记代表给定芯片的数据结构。同时，从数据结构 amd7930_ops 中也约略可以看出对 AMD7930 芯片的操作有：打开/关断输入或输出、控制输入或输出的音量、控制监听音量、设置输入或输出信道的各种参数和对电平的分辨率、设置数字化及还原时采用的编码制式以及采样的频率等等。应用程序通常通过系统调用 ioctl( )来启动这些函数。

函数 register_sparcaudio_driver( )是由上层模块 sparcaudio 移出，供下层模块（如 amd7930）用来向它登记的。除此以外，有些需要在上层模块中为下层模块完成的操作，如某些资源的分配，也可以放在这个函数中进行（当然，也可以由下层模块自己完成）。所以，这个函数所做的事情并不只是登记下层模块，还包括为具体音频信道（即 amd7930 芯片）设置缓冲区等，所以代码比较长，我们不在这里列出其代码了。所谓登记，在这里主要是将一个 sparcaudio_driver 结构指针传递给 sparcaudio，使它可以通过这个数据结构访问具体 amd7930 信道的一些数据，并且可以通过结构中的指针 ops 找到对 amd7930 操作的各种函数指针。在 sparcaudio 模块中有个 sparcaudio_driver 结构指针数组，用来保存已向它登记的指针，从而维持与下层模块的联系，此数组的定义在 audio.c 中给出：

```
73    static struct sparcaudio_driver *drivers[SPARCAUDIO_MAX_DEVICES];
```

相应地，在 amd7930 模块中则有一个 sparcaudio_driver 结构数组，也叫 driver[ ]，定义于 amd7930.c 中：

```
118    static struct sparcaudio_driver drivers[MAX_DRIVERS];
```

这两个数组虽然同名，但是分属两个不同的模块。两个模块都是 static，所以即使把两个模块都作为静态模块编译并且连接，也不会造成冲突。这里要指出，sparcaudio 中虽然有个 sparcaudio_driver 结构指针数组，这并不意味着系统中要安装许多个 amd7930 的模块，而是意味着系统中可以有许多个 amd7930 芯片。所以，amd7930 模块只有一个，但是它操作的对象，或者说操作的上下文，却可以有多个，而数组中的指针则指向由同一模块驱动的各个对象。

除登记 sparcaudio_driver 结构指针以外，register_sparcaudio_driver( )中还有件重要的事情，那就是为每个 amd7930 芯片在文件系统中创建文件节点，使应用程序可以"看见"这个具体的设备。就 devfs 设备文件系统而言，这是通过 devfs_register( )完成的，在由 sparcaudio 模块所创建的目录"sound"下为每个 amd7930 芯片创建文件节点，如"sound/audio1"、"sound/audio2"等等。实际上，amd7930 是个很复杂的芯片，根据其结构可以进一步划分成混音、数字信号处理、音频控制/状态等功能模块；所

以在 sparcaudio 模块的设计中选择将次设备号编码表示不同的功能模块，而为每一个功能模块都创建一个文件节点，如 sound/audio1、sound/mixer1 以及 sound/dsp1 等等。当然，也可以选择将它们合在一起。于是，每个这样的文件节点都对应着一个次设备号，而每个路径名则对应着主设备号与次设备号的一种组合。

总之，每个设备驱动模块在安装后都要使它的上层模块能"看见"它，有时还要使应用程序也能"看见"它。前者通过向上层模块登记而实现，后者则通过 mknod( )、devfs_mk_dir( )、devfs_register( ) 一类的函数实现；并且既可以由下层模块自己完成，也可由其上层模块替它完成，就如这里 sparcaudio 替 amd7930 创建文件节点那样。实际上，后者在本质上也是一种登记的过程，例如 mknod( )就可以看成是由内核提供、让各种模块向文件系统登记的函数。通常一个模块堆栈中至少有一个模块需要为应用程序所见。

作为实例，我们再看一下 amd7930 模块的 cleanup_module( )：

```
1713    #ifdef MODULE
1714    void cleanup module(void)
1715    {
1716        register int i;
1717
1718        for (i = 0; i < num_drivers; i++) {
1719            amd7930_detach(&drivers[i]);
1720            num_drivers--;
1721        }
1722    }
1723    #endif
```

显然，amd76930_detach( )是 amd7930_attach( )的逆操作：

[cleanup_module( ) > amd76930_detach( )]

```
1662    #ifdef MODULE
1663    /* Detach from an amd7930 chip given the device structure. */
1664    static void amd7930_detach(struct sparcaudio_driver *drv)
1665    {
1666        struct amd7930_info *info = (struct amd7930_info *)drv->private;
1667
1668        unregister_sparcaudio driver(drv, 1);
1669        amd7930 idle(info);
1670        disable irq(info->irq);
1671        free_irq(info->irq, drv);
1672        sbus_iounmap(info->regs, info->regs_size);
1673        kfree(drv >private);
1674    }
1675    #endif
```

这里 unregister_sparcaudio_driver( )是由 sparcaudio 模块移出、供下层模块撤销登记用的。可想而知，

它从 devfs 文件系统中删去相应的文件节点，释放有关的缓冲区，并从 sparcaudio 的指针数组 drivers[ ] 中将相应的表项恢复成 NULL。撤销向上层的登记以后，还要将有关硬件设置成“空转”状态，并与中断向量脱钩、以及撤销硬件寄存器的内存映象。总之，要消除 amd7930_attach( )的所有影响，使系统恢复到相应的 amd7930_attach( )操作之前的状态。

可安装模块的作用并不局限于设备驱动程序的实现，对作为可选项的许多文件系统（格式）的支持就常常是通过可安装模块实现的。此外，socket 机制也可以选择通过可安装模块来实现。至于各种网络规程的实现，那就更是非可安装模块莫属了。当然，网络规程的实现也可以看作是广义的设备驱动程序。另一方面，模块在应用程序界面上的表现（即文件节点）也并不一定在/dev 目录下或 devfs 子树中，如代表着插口的文件节点就通常不在这些目录中。

一个模块也并不限于只由一个文件节点代表，许多模块通过 create_proc_read_entry( )在/proc 子树中也创建一个只读的文件节点，供用户或应用程序读取有关的状态和统计信息。更进一步，可安装模块甚至可以根本不与/dev 目录和 devfs 打交道，也不使用主设备号和次设备号，甚至不向上层模块登记，而只是通过 proc_register( )在/proc 文件系统中创建一个文件节点，内核通过/proc 文件系统就可以访问到具体的模块。实际上，proc_register( )是将向上层模块(/proc 文件系统)和向应用程序界面的登记(/proc 子树中的文件节点）合二为一了，所以只适合用于高层模块或一共只有一层的模块，总之是直接与 vfs 层接口的模块。这个函数既可以在/proc 文件系统中创建文件节点，也可以创建目录节点和符号连接节点，因此可以用来在/proc 目录下建立起类似于 devfs 的多层结构。由于在这种文件节点中并不使用主设备号和次设备号，就特别适用于一些高层的，严格说来算不上设备驱动的模块（如高层的网络规程)，以及一些不适合（或有困难）为之分配主 / 次设备号的模块。缺点是它只支持 read( )、write( )和 lseek( ) 三种文件操作，函数 proc_register( )的代码在 fs/proc/generic.c 中：

```
350   static int proc_register(struct proc_dir_entry * dir,
                                        struct proc_dir_entry * dp)
351   {
352       int i;
353
354       i = make_inode_number( );
355       if (i < 0)
356           return -EAGAIN;
357       dp->low_ino = i;
358       dp->next = dir->subdir;
359       dp->parent = dir;
360       dir->subdir = dp;
361       if (S_ISDIR(dp->mode)) {
362           if (dp->proc_iops == NULL) {
363               dp->proc_fops = &proc_dir_operations;
364               dp->proc_iops = &proc_dir_inode_operations;
365           }
366           dir->nlink++;
367       } else if (S_ISLNK(dp->mode)) {
368           if (dp->proc_iops == NULL)
369               dp->proc_iops = &proc_link_inode_operations;
370       } else if (S_ISREG(dp->mode)) {
```

```
371            if (dp->proc_fops == NULL)
372                dp->proc_fops = &proc_file_operations;
373        }
374        return 0;
375    }
```

参数 dir 指向父目录的 proc_dir_entry 数据结构，如果父目录就是/proc 则为&proc_root。另一个参数 dp，则指向待创建节点的 proc_dir_entry 结构。在/proc 文件系统中不管是目录节点还是文件节点都使用同一种数据结构，即 proc_dir_entry，并且该结构中包含了通常分布在 dentry 和 inode 两种数据结构中的信息。读者可回顾一下"/proc 特殊文件系统"一节中的有关内容。在调用 proc_register( )之前，可安装模块要分配并初步设置好一个 proc_dir_entry 数据结构。这个数据结构中有三个函数指针 read_proc、write_proc 以及 get_info，应分别指向模块中的相应函数，它们的界面定义于 include/linux/proc_fs.h:

```
47    typedef int (read_proc_t)(char *page, char **start, off_t off,
48                    int count, int *eof, void *data);
49    typedef int (write_proc_t)(struct file *file, const char *buffer,
50                    unsigned long count, void *data);
51    typedef int (get info_t)(char *, char **, off_t, int);
```

其中 get_info 也是用于读文件的，与 read_proc 的不同之处仅在于函数的调用界面。相比之下，read_proc 多了最后两个参数，一个是指针 eof，用来返回表示文件是否已经读到了结尾，另一个指针 data 就是 proc_dir_entry 结构中的指针 data，可以用来传递一些与特定数据结构有关的信息，例如类似于次设备号那样的数据结构序号。不过，这两个函数中只能选择其一，用了 read_proc 就不用 get_info。

在函数 proc_register( )中，根据待创建节点的性质（模式）而设置其 proc_dir_entry 结构中的若干指针。如果是目录节点就设置其 proc_fops 和 proc_iop 两个指针，使它们分别指向/proc 文件系统的 inode_operations 数据结构和用于目录节点的 file_operations 数据结构；如果是符号连接节点就使 proc_iops 指向/proc 用于符号连接的 inode_operations 数据结构；而如果是文件节点则使 proc_fops 指向/proc 用于文件节点的 file_operations 数据结构。这几个数据结构都是/proc 文件系统所固有的。前三个数据结构中的函数指针保证了当应用程序通过系统调用 open( )在内核中启动 path_walk( )时能找到/proc 文件系统中的节点，而最后一个数据结构中的函数指针，则提供了进入具体模块进行读 / 写的"中转站"。当打开一个/proc 文件时，内核会在 proc_lookup( )中通过 proc_get_inode( )为目标节点创建 inode 数据结构，把指向这个数据结构的指针 proc_fops 复制到 inode 结构中（指针 i_fop），然后又复制到 file 结构中（指针 f_op）。

我们来看看这个数据结构，这是在 fs/proc/generic.c 中定义的：

```
36    static struct file_operations proc_file_operations = {
37        llseek:       proc_file_lseek,
38        read:         proc_file_read,
39        write:        proc_file_write,
40    };
```

从这个结构中可以看出，这个机制仅支持 llseek、read 和 write 三种文件操作，而不支持 ioctl。在设备驱动程序中，ioctl 是个很灵活、容量很大，因而功能很强、很重要的操作。例如，在前面的 amd7930 模块中定义了那么多的低层操作，而其离开了对 ioctl( )系统调用的支持就根本无法实现。因此，缺少对 ioctl 的支持是这个机制的一个缺点。当然，以后在更新的版本中也许会把这一点考虑进去（再说那也是很容易的事）。

其次，结构中的函数指针都指向由/proc 文件系统提供的通用性程序，而不是指向具体模块中的有关函数。以读操作为例，跳转时的第一站为 proc_file_read( )，其代码在同一文件（generic.c）中：

[sys_read( ) > proc_file_read( )]

```
49    static ssize_t
50    proc_file_read(struct file * file, char * buf, size_t nbytes, loff_t *ppos)
51    {
52        struct inode * inode = file->f_dentry->d_inode;
53        char    *page;
54        ssize_t retval=0;
55        int eof=0;
56        ssize_t n, count;
57        char    *start;
58        struct proc_dir_entry * dp;
59
60        dp = (struct proc_dir_entry *) inode->u.generic_ip;
61        if (!(page = (char*) __get_free_page(GFP_KERNEL)))
62            return -ENOMEM;
63
64        while ((nbytes > 0) && !eof)
65        {
66            count = MIN(PROC_BLOCK_SIZE, nbytes);
67
68            start = NULL;
69            if (dp->get_info) {
70                /*
71                 * Handle backwards compatibility with the old net
72                 * routines.
73                 */
74                n = dp->get_info(page, &start, *ppos, count);
75                if (n < count)
76                    eof = 1;
77            } else if (dp->read_proc) {
78                n = dp->read_proc(page, &start, *ppos,
79                            count, &eof, dp->data);
80            } else
81                break;
82
83            if (!start) {
84                /*
```

```
 85                * For proc files that are less than 4k
 86                */
 87               start = page + *ppos;
 88               n -= *ppos;
 89               if (n <= 0)
 90                   break;
 91               if (n > count)
 92                   n = count;
 93           }
 94           if (n == 0)
 95               break;  /* End of file */
 96           if (n < 0) {
 97               if (retval == 0)
 98                   retval = n;
 99               break;
100           }
101
102           /* This is a hack to allow mangling of file pos independent
103            * of actual bytes read.  Simply place the data at page,
104            * return the bytes, and set `start' to the desired offset
105            * as an unsigned int. - Paul.Russell@rustcorp.com.au
106            */
107           n -= copy_to_user(buf, start < page ? page : start, n);
108           if (n == 0) {
109               if (retval == 0)
110                   retval = -EFAULT;
111               break;
112           }
113
114           *ppos += start < page ? (long)start : n; /* Move down the file */
115           nbytes -= n;
116           buf += n;
117           retval += n;
118       }
119       free_page((unsigned long) page);
120       return retval;
121   }
```

在第 69～80 行中，根据函数指针 get_info 和 read_proc 是否为 NULL 而调用其中之一（或二者都不调用），这才进入了由模块所提供的函数中。经过这样一层中转，程序的执行效率多少受到一点影响。相比之下，如果模块直接向内核登记其 file_operations 数据结构（像 sparcaudio 那样），则在经由 file_operations 结构跳转时直接就进入了由模块提供的函数。

但是，尽管如此，将文件节点创建在/proc 文件系统中还是很有吸引力的。摆脱了对主/次设备号的依赖，就使可安装模块的设计变得更灵活，并因不再有设备号惟一性的问题而更可移植，程序设计也要简单一些。

那么，是否可以像对待普通文件那样，把代表可安装模块的文件节点创建在文件系统中任意（除

/proc 文件以外）的位置上呢？如果不使用主/次设备号的话，至少在目前的内核上是不行的。这里的困难在于 file_operations 数据结构是每个文件系统（类型）一个，而不是每个文件一个（devfs 文件系统是个例外），再说可安装模块也并不是一种特殊的文件类型。当然，如果仍旧使用主设备号/次设备号，那是可以的。可正是因为使用设备号才更有必要把这些文件集中在一起，因为那样才便于观察到设备号的冲突。

8.4 PCI 总线

说到外部设备的驱动就离不开系统的外设总线。最初 PC 机中的外设总线只是 8 位的，就是说每次只能读/写一个字节；后来扩充成 16 位，称为 ISA 总线；再后来又扩充成 32 位的 EISA 总线。但是，随着技术的发展和应用的日益普及，人们逐渐认识到这些总线都存在着一些重要的、根本性的缺点，因而需要开发出一种全新的总线。当时提出了两种主要的候选总线结构，一种称为 VESA，另一种就是 PCI。经过一段时期的竞争，PCI 总线成为了事实上的标准总线，不光是 PC 系统结构中的标准总线，也是许多其他系统结构中的标准总线。现在的 PC 机一般都提供若干 PCI 总线插槽，同时也提供少量 ISA 总线插槽以求跟旧式接口卡的兼容。至于 EISA，则还来不及推广就被 PCI 取代了。以前，外设总线一般都是与具体的 CPU 和系统结构密切联系的，CPU 不同、系统结构不同，采用的总线也就不同。例如 ISA 总线就是用于采用 Intel X86 系列 CPU 的 PC 中，UNIBUS 和 Qbus 就是用于以前的 PDP-11 中，而 VME 总线则用于 Motorola 的 M68K 和 Power PC 系列的系统中。这一点可以说是历来如此。可是，自从 PCI 总线问世以后，却很快就成为了通用的标准总线，以至于不管是什么 CPU，不管是为哪种外设开发的芯片组，都会提供跟 PCI 总线的接口。从这个意义上说，似乎 PCI 总线成了计算机系统结构的中心，而 CPU 倒反而退居从属的地位了。随着多处理器 SMP 结构的采用和普及，这种趋向就更明显了，因为系统中可以有许多个处理器，而 PCI 总线却往往只有一条。然而，PCI 总线的标准，即其规格书，是个不太好读、不太好理解的文本。拿到一块 PCI 接口卡或者一组芯片，要从规格书或厂商提供的说明书出发，开发出该项设备的驱动程序实非易事。在这个意义上，Linux 内核中有关的源代码恰恰为我们深入理解和运用 PCI 总线(及设备)提供了一个实例、一个样板。

以后，为行文的方便，在不至于引进混淆的场合我们将直接以“总线”或“PCI”表示“PCI 总线”。

那么，ISA 总线(以及 PCI 以前的那些总线)到底有些什么缺点呢？换言之，PCI 总线有些什么优点呢？

首先，ISA 乃至 EISA 的速度都太慢。这两种总线的时钟频率都是 8.33MHz，就算是 32 位的 EISA，其理论上的最大通量也不过是每秒 33MB(ISA 的最大通量只是 8.33MB)。这显然不能满足图像、网络等方面的应用需要。例如，光是一个 100 兆的 Ethernet 接口，其理论上的最大通量就已经超过 12MB 了。但是，提高总线的速度并不单纯是个提高时钟频率的问题。在高速的条件下，有不少物理问题(如电信号的传输)要考虑和解决。而且，如果采用相同的插槽则还要考虑跟已经存在的接口卡兼容的问题。所以，PCI 总线采用了完全不同的插槽，时钟频率则提高到 33MHz，使理论上的最大通量提高到 133MB。而且，还为进一步把时钟频率提高到 66MHz、总线宽度提高到 64 位留下了余地。

其次是地址的分配与设置。就像存储器本身是一种资源一样，存储器的地址也是一种资源。在同一时间内，一个地址只能惟一地用于一个物理的存储单元，或者就空闲不用。在 i386 系统结构中，对内存的访问和对输入/输出寄存器的访问通过两套不同的指令完成，所以有存储器和 I/O 两个不同的地

址空间。一般而言，内存的物理地址以及输入/输出寄存器的地址是由硬件决定的，不过对于内存的物理地址还可以通过地址映射机制来一次转换。可是，怎样处理外部设备上的存储空间呢？ISA (以及其他早期的总线) 接口卡上都采用一些跳线或小开关，将接口卡插上总线前都要先通过这些小开关设置好地址(还有中断请求号)，而对于软件则也要在一个“安装”过程中加以设置，使二者相符。可是，这样做不但麻烦，还往往成为系统因为地址冲突或不符而不能正常运行的原因。再说，随着外部设备的日益复杂，这样做已经不大现实了。所以，理想的办法是由系统软件自动设置，总的思路是：

(1) 每块接口卡(每项外设)都通过某种途径告诉系统：卡上有几个存储区间及 I/O 地址区间，每个区间是多大，以及各自在卡上(本地)的地址。这些地址在本质上都是局部的、内部的，所以都从 0 起算。但是，这些区间不与总线直接相连，在把接口卡插上总线并加电之初，从总线上还访问不到这些区间，所以不会互相冲突。为区别这种地址，我们不妨称之为“卡上地址”，虽然实际上未必是在接口卡上，有些外设芯片其实是固定在母板上的。

(2) 系统软件在知道了一共有多少外部设备、各自又有什么样的存储区间以后，就可以统筹地为这些区间分配“物理地址”，并且建立起这些区间与总线之间的连接，以后就可以通过这些地址来访问。显然，这里所谓“物理地址”与真正的物理地址是有些区别的，它实际上也是一种逻辑地址，所以常称为“总线地址”，因为这是 CPU 在总线上所看到的地址。可想而知，接口卡上一定有着某种地址映射机制。所谓“为外设分配地址”，就是为其分配总线地址，并为之建立起映射。这种映射代替了从前的跳线或小开关，卡上有几个地址区间就有几个映射。对于 CPU，“总线地址”就相当于物理地址，还可以通过虚存地址的映射再加一次变换。此外，对于中断请求线的连接也与此相似。

事实上，PCI 总线正是这样设计的。此外，对于 I/O 地址空间与内存地址空间相分离的系统结构，如 i386 结构，还可以选择将 I/O 寄存器的地址也映射成内存地址(总线地址)，这样就可以通过访内指令来操作这些寄存器了。这一方面有可能简化程序设计，另一方面也可避免 i386 系统结构中 I/O 地址空间太小(16 位)、太拥挤的问题。

可以想像，每个 PCI 设备或者接口卡上一定有许多用来完成这个过程，即建立起这些连接和映射的寄存器，系统或设备初始化的时候要通过这些寄存器来“配置”该设备的各个总线地址区间。可是，这些寄存器本身的地址又怎么办呢？这不是又回到了原先的问题吗？下面读者就会看到，PCI 总线的设计者对这个问题解决得很好（但是从程序设计的角度却有些令人头痛)。

第三个问题是对使用总线的竞争。在早期的计算机系统中，整个系统只有一个 CPU，从总线的角度来说，这个 CPU 就是“主设备”(Master)，而其他的都是“从设备”，只有主设备才能启动跨总线的操作。需要由“从设备”启动的操作也是有的，那就是对内存的 DMA 操作。此时“从设备”先向 CPU 发出一个 DMA 请求，让 CPU 暂停访问内存，实际上是暂停包括访问总线在内的一切外部操作，使系统中暂时没有了“主设备”。得到允许以后，这“从设备”就暂时升级变成了对内存的“主设备”，从而可以启动内存操作。这种内存操作当然是跨总线的，但是因为 CPU 已暂停活动，所以不存在竞争使用总线的问题。或者说，对(使用)总线的竞争寓于对(使用)内存的竞争之中。可是，在多处理器的系统中，对总线的竞争就成为问题了，如果两个 CPU 同时启动跨总线的访问，怎样来解决冲突呢？还有，随着技术的发展，一些“从设备”，即外设接口卡也带上了智能，有了本地的处理器，在这样的情况下，应该允许一个“从设备”直接访问另一个“从设备”上的存储区间或 I/O 寄存器，而不必由 CPU 介入。这与 DMA，即“(由外设)直接访问内存”，是同样的概念。可是，如果一个“从设备”要直接访问另一

个“从设备”，那当然也要先取得对总线的使用权，暂时变成总线的主设备。为了解决竞争使用总线的问题，PCI 总线上配备了一个仲裁器。遇有冲突时，仲裁器会选择其中之一(包括 CPU)暂时成为当前的“主设备”，而其他的则只好等待。由此又生出另一个问题来，如果 CPU，或一个设备，想要启动的是写操作，但是由于冲突而一时不能成为总线的“主设备”，那么它是只能停下来等待；还是可以让它把要写的内容放在一个缓冲区中，把它托付给 PCI 总线，自己则接着继续运行？显然这对于 CPU 的效率是有影响的。PCI 总线的设计考虑到了这个问题，为写操作提供了缓冲。当然，对于读操作就没有办法，只好等待了。所以，从效果上看，跨 PCI 总线的写操作往往比读操作快。可想而知，在这个方面，PCI 总线的硬件(芯片)设计是相当复杂的，但好在对软件是“透明”的，所以在代码中看不到这个问题的存在。但是，允许“从设备”通过竞争暂时变成“主设备”，则实际上是对 DMA 的推广，因而对外部设备接口的设计与实现，进而对设备驱动程序的代码有着重要的影响。我们将在“块设备驱动”一节中结合 DMA 操作进一步讨论这个问题。

还有个问题是总线的扩充问题。一块母板上能容纳的插槽、或不经插槽直接与 PCI 总线相连的芯片的数量总是有限的，能不能在需要时通过一块接口卡对总线的容量加以扩充，或者连接上另一条(同种或异种)总线呢？读者也许觉得这个问题很简单，其实不然(不过我们在这里就不深入讨论了)。PCI 总线在这方面也解决得很好。人们设计出了各种各样的“PCI 桥”芯片，通过一个 PCI 桥就可以连接到一条 PCI 总线。CPU 通过“宿主—PCI 桥”与一条 PCI 总线相连，处在这种位置上的 PCI 总线称为“主(Primary)PCI 总线”，或者就称“主总线”。PC 机中通常只有一个“宿主—PCI 桥”，但是在特殊的系统结构中也可以有多个。在一条 PCI 总线的基础上，可以再通过“PCI 桥”连接到其他次一层的总线，例如通过“PCI-PCI 桥”可以连接到另一条 PCI 总线，通过“PCI-ISA 桥”可以连接到一条 ISA 总线。事实上，现代 PC 机中的 ISA 总线正是通过“PCI-ISA 桥”连接在 PCI 总线上的。这样，通过使用“PCI-PCI 桥”，就可以构筑起一个层次的、树状的 PCI 系统结构。对于上层的总线而言，连接在这条总线上的 PCI 桥也是一个设备。但是，这是一种特殊的设备，它既是上层总线上的一个设备，实际上又是上层总线的延伸。

实际上，在 PC 机中，“宿主—PCI 桥”与内存控制器往往都做在同一个芯片中。CPU 与这个器件、以及通过这个器件与内存的连接就是原来意义上的“系统总线”，而通过这个器件直接相连的就是系统的主 PCI 总线。所有的外设，包括磁盘、键盘、鼠标器、并行口、串行口、网络卡、等等，全都直接或间接地接在 PCI 总线上。其中有些以接口卡和插槽的形式相连，有的则固定在系统母板上通过电子线路直接相连。特别地，PC 机中一般还通过一个“PCI-ISA 桥”连接到一条 ISA 总线，提供若干 ISA 插槽。不过，插在 ISA 总线上的接口卡就不具备 PCI 的地址映射功能了。此外，如前所述，需要时还可以通过“PCI-PCI 桥”连接到其他 PCI 总线。这样，如果把 CPU 比喻作一个城市的中心，系统总线就好像是“一环路”，主 PCI 总线就好像是“二环路”，而 ISA 总线以及连在主 PCI 总线上的其他 PCI 总线就是“三环路”了。图 8.2 所示是个 PC 系统结构的示意图。

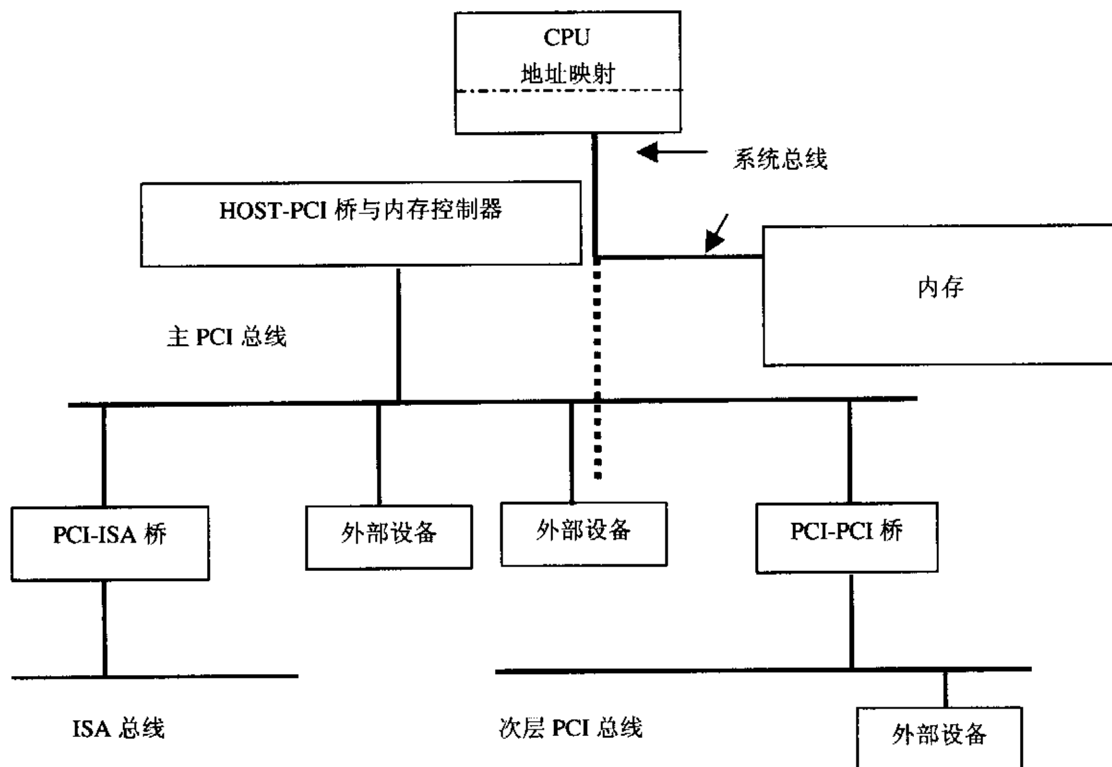


图 8.2　PC 系统结构与 PCI 总线

图 8.2 中在系统总线与 PCI 总线上的一个设备之间有条虚线，表示 PCI 总线上的设备在完成了配置以后就好像连接在系统总线上一样。而在此之前则对于系统总线和 PCI 总线都是不可见的，所以不会互相冲突。对于连接在次层 PCI 总线上的设备也是一样。此外，系统的内存控制器与 HOST-PCI 桥通常都集成在同一芯片中，可以把内存也看作是连在 PCI 总线上的一个设备，不过它永远是“从设备”。

从上面的介绍可以看出，PCI 总线的设计考虑到了种种方面的问题，并且解决得很好，它之所以能成为受到普遍接受的标准总线绝不是偶然的。

前面讲过，每个 PCI 设备或者接口卡上有许多用于地址配置的寄存器，初始化时要通过这些寄存器来“配置”该设备的总线地址。一旦完成了配置以后，CPU 就可以访问该设备的各项资源，就好像那是内存的一部分一样，这就没有什么特殊的了。所以，本节的重点是配置 PCI 设备的过程，与这个过程有关的代码并不简单。在本书中，我们把每个设备上的这些寄存器合在一起称为该设备的“配置寄存器组”。

PCI 标准规定每个设备的配置寄存器组最多可以有 256 字节的连续空间，其中开头 64 个字节的用途和格式是标准的，称为配置寄存器组的“头部”(configuration header)。这样的“头部”又有两种，其中“0 型”(type 0)头部用于一般的 PCI 设备，“1 型”头部则用于各种 PCI 桥。但是，不管是“0 型”还是“1 型”，其开头 16 个字节的用途和格式总是共同的。这 16 个字节中包含着有关头部的类型、设备的种类、设备的一些性质、由谁制造等等信息。对于这 16 个字节的地址，include/linux/pci.h 中定义了这样一些常数：

```
24      #define PCI_VENDOR_ID         0x00    /* 16 bits */
25      #define PCI_DEVICE_ID         0x02    /* 16 bits */
26      #define PCI_COMMAND           0x04    /* 16 bits */
        ......
38      #define PCI_STATUS            0x06    /* 16 bits */
        ......
54      #define PCI_CLASS_REVISION    0x08    /* High 24 bits are class, low 8
55                                                                  revision */
56      #define PCI_REVISION_ID       0x08    /* Revision ID */
57      #define PCI_CLASS_PROG        0x09    /* Reg. Level Programming Interface */
58      #define PCI_CLASS_DEVICE      0x0a    /* Device class */
59
60      #define PCI_CACHE_LINE_SIZE   0x0c    /* 8 bits */
61      #define PCI_LATENCY_TIMER     0x0d    /* 8 bits */
62      #define PCI_HEADER_TYPE       0x0e    /* 8 bits */
```

这里的寄存器 PCI_HEADER_TYPE 表明是哪一种头部，PCI_CLASS_DEVICE 和 PCI_CLASS_PROG 则表明是哪一种设备。例如，PCI_CLASS_DEVICE 的高 8 位为 0x02 表示是网络设备，而低 8 位为 0 且 PCI_CLASS_PROG 为 0，又进一步表示这是 Ethernet 接口卡。又如，PCI_CLASS_DEVICE 的高 8 位为 0x07 表示是“简单通信控制器”，低 8 位为 01 进一步表示是并行口，而 PCI_CLASS_PROG 为 0 表示单向、为 1 表示双向、为 2 表示符合 ECP 1.0 规定(IEEE1284 并行口标准中的一种操作模式)。还有，PCI_VENDOR_ID 表示由哪一家厂商制造，如 Intel 的 ID 号码是 0x8086、Compaq 的 ID 号码是 0x0e11。这些号码是由一个统一的机构分配、指定的，所以不会重复。每家厂商对其产品也有编号，那就是寄存器 PCI_DEVICE_ID 中的内容。所有这些信息都是由厂商固化在产品中的，不会变化。在 Linux 系统上，可以通过“cat /proc/pci”查看系统中所有 PCI 设备的类别、型号以及厂商等等信息，那就是从这些寄存器来的。

所以，根据这些信息就可以确定应该怎样进一步解释和处理其余的 48 个字节。至于头部以外的 192 个字节，则取决于具体的设备，如果不需要也可以没有。除地址配置以外，这些(头部)寄存器还有个重要的作用，就是使 CPU 能够探测到相应设备(接口)的存在，并且确定该设备的种类和一些特性，包括由谁制造等等。这样，用户就不再需要通过种种途径告知系统都有哪一些外设，而改由 CPU 通过一个称为“枚举”的过程自动扫描探测所有连接在 PCI 总线上的外设。

如前所述，这些寄存器本身的地址就是一个问题。首先，很难为不同的设备指定不同的起始地址，因为那样就又回到了原来的问题上。试想，每个设备都可以有 256 字节的配置寄存器组，如果要为每一种可能的设备都保留不同的地址，那么 1024 种设备就得保留 256KB，一万种呢？更多呢？何况这又给地址的管理带来了麻烦，这显然是不现实的。比较好的办法是让所有设备的配置寄存器组都采用相同的地址，由所在总线的 PCI 桥在访问时附加上其他条件来区分。而 CPU 则通过一个统一的入口地址向“宿主—PCI 桥”发出命令，由相应的 PCI 桥间接地完成具体的读写。对于 i386 结构的处理器，PCI 总线的设计者在 I/O 地址空间保留了 8 个字节用于这个目的，那就是 0xCF8～0xCFF。这 8 个字节实际上构成两个 32 位的寄存器，第一个是“地址寄存器”0xCF8，第二个是“数据寄存器”0xCFC。要访问某个设备中的某个配置寄存器时，CPU 先往地址寄存器中写入目标地址，然后通过数据寄存器读写数据。不过，写入地址寄存器中的目标地址是一种包括总线号、设备号、功能号以及配置寄存器地址

在内的综合地址，其构成如图 8.3 所示。

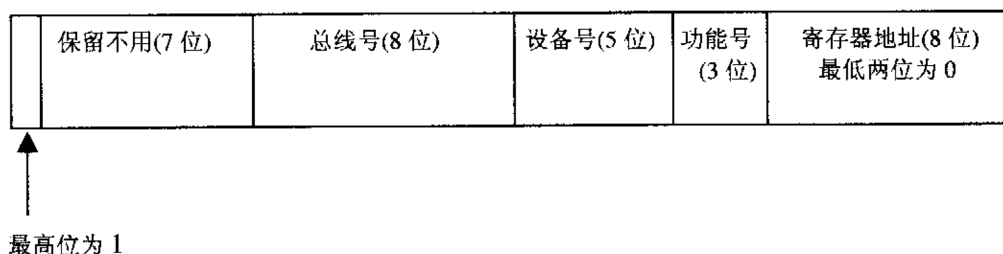


图 8.3 写入地址寄存器 0xCF8 的综合地址

这里的总线号、设备号以及功能号是对配置寄存器地址的扩充，就是上面说的附加条件。首先，每条 PCI 总线都有个总线号，主总线的总线号固定为 0，其余的则由 CPU 在枚举阶段每当探测到一个 PCI 桥时便为其指定一个，依次递增。设备号一般代表着一块 PCI 接口卡(更确切地说是 PCI 总线接口芯片)，通常取决于插槽的位置。每块 PCI 接口卡上可以有若干个功能模块，这些功能模块共用同一个 PCI 总线接口芯片，包括其中用于地址映射的电子线路，以降低成本。从逻辑的角度说，每个“功能”实际上就是一项设备，所以设备号和功能号合在一起又可以称作“逻辑设备号”，而每块卡上最多可以容纳 8 个逻辑设备。显然，这些字段结合在一起就惟一地确定了系统中的一项 PCI 逻辑设备。读者也许会问：一开始的时候，在尚未为各条总线指定总线号之前，又怎样来访问特定的总线呢？事实上，一开始的时候只有 0 号总线是可访问的，在扫描 0 号总线时如果发现上面某一个设备是 PCI 桥，就为之指定一个新的总线号，例如 1，这样 1 号总线也就可以访问了，这就是枚举阶段的任务之一。

一般，每个逻辑设备中都有几个需要映射的地址区间，所以在 0 型头部中定义了 6 个“基地址”寄存器，可以分别用于 6 个地址区间的映射。下面读者就会看到，对这些寄存器的操作是相当复杂的。例如，直接读时是区间的地址加上一些标志位，而若先往里面写上全 1 再读就成了区间长度。除了这 6 个常规的区间外，逻辑设备中有时候还会有一块 ROM，所以又有一个“扩充 ROM 基地址”寄存器。此外，还有用来设置中断请求线的寄存器以及其他几个寄存器。

对于 0 型头部中这些寄存器的地址，include/linux/pci.h 中定义了一些常数：

```
72    /*
73     * Base addresses specify locations in memory or I/O space.
74     * Decoded size can be determined by writing a value of
75     * 0xffffffff to the register, and reading it back.  Only
76     * 1 bits are decoded.
77     */
78    #define PCI_BASE_ADDRESS_0  0x10    /* 32 bits */
79    #define PCI_BASE_ADDRESS_1  0x14    /* 32 bits [htype 0,1 only] */
80    #define PCI_BASE_ADDRESS_2  0x18    /* 32 bits [htype 0 only] */
81    #define PCI_BASE_ADDRESS_3  0x1c    /* 32 bits */
82    #define PCI_BASE_ADDRESS_4  0x20    /* 32 bits */
83    #define PCI_BASE_ADDRESS_5  0x24    /* 32 bits */
```

```
      ......
96    /* Header type 0 (normal devices) */
97    #define PCI_CARDBUS_CIS        0x28
98    #define PCI_SUBSYSTEM_VENDOR_ID 0x2c
99    #define PCI_SUBSYSTEM_ID       0x2e
100   #define PCI_ROM_ADDRESS        0x30    /* Bits 31..11 are address, 10..1 reserved */
      ......
104   #define PCI_CAPABILITY_LIST 0x34    /* Offset of first capability list entry */
105
106   /* 0x35-0x3b are reserved */
107   #define PCI_INTERRUPT_LINE  0x3c    /* 8 bits */
108   #define PCI_INTERRUPT_PIN   0x3d    /* 8 bits */
109   #define PCI_MIN_GNT         0x3e    /* 8 bits */
110   #define PCI_MAX_LAT         0x3f    /* 8 bits */
```

以后，随着代码的阅读，读者自会明白这些寄存器的作用和有关的操作。

PCI 桥的作用和功能不同于一般设备，1 型头部中的寄存器自然也就不同。对于 1 型头部中的寄存器地址，include/linux/pci.h 中另有一些常数定义：

```
112   /* Header type 1 (PCI-to-PCI bridges) */
113   #define PCI_PRIMARY_BUS           0x18    /* Primary bus number */
114   #define PCI_SECONDARY_BUS         0x19    /* Secondary bus number */
115   #define PCI_SUBORDINATE_BUS       0x1a    /* Highest bus number behind the bridge */
116   #define PCI_SEC_LATENCY_TIMER     0x1b    /* Latency timer for secondary interface */
117   #define PCI_IO_BASE               0x1c    /* I/O range behind the bridge */
118   #define PCI_IO_LIMIT              0x1d
      ......
123   #define PCI_SEC_STATUS            0x1e    /* Secondary status register,
                                                       only bit 14 used */
124   #define PCI_MEMORY_BASE           0x20    /* Memory range behind */
125   #define PCI_MEMORY_LIMIT          0x22
      ......
128   #define PCI_PREF_MEMORY_BASE      0x24    /* Prefetchable memory range behind */
129   #define PCI_PREF_MEMORY_LIMIT     0x26
      ......
134   #define PCI_PREF_BASE_UPPER32     0x28    /* Upper half of prefetchable
                                                              memory range */
135   #define PCI_PREF_LIMIT_UPPER32    0x2c
136   #define PCI_IO_BASE_UPPER16       0x30    /* Upper half of I/O addresses */
137   #define PCI_IO_LIMIT_UPPER16      0x32
138   /* 0x34 same as for htype 0 */
139   /* 0x35-0x3b is reserved */
140   #define PCI_ROM_ADDRESS1     0x38    /* Same as PCI_ROM_ADDRESS, but for htype 1 */
141   /* 0x3c-0x3d are same as for htype 0 */
142   #define PCI_BRIDGE_CONTROL        0x3e
```

在 1 型头部中也有两个地址区间，所以前面定义的寄存器 PCI_BASE_ADDRESS_0 和 PCI_BASE_ADDRESS_1（见 78 和 79 行）也适用于 1 型头部。PCI 桥跨在两条总线之间，寄存器 PCI_PRIMARY_BUS 和 PCI_SECONDARY_BUS 的内容就说明了其上下两端的总线号，其中 PCI_SECONDARY_BUS 就是该 PCI 桥所连接和控制的总线，而 PCI_SUBORDINATE_BUS 则说明自此以下、在以此为根的子树中最大的总线号是什么。CPU 在枚举阶段所作的是深度优先的扫描，所以子树中的总线号总是连续递增的。当 CPU 往 I/O 寄存器 0xCF8 中写入一个综合地址以后，从 0 号总线开始，每个 PCI 桥会把综合地址中的总线号与其自身的总线号相比，如果相符就用逻辑设备号在本总线上寻访目标设备；否则就进一步把这个总线号与 PCI_SUBORDINATE_BUS 中的内容相比，如果目标总线号落在当前子树的范围中，就把综合地址传递给其下的各个次层 PCI 桥，要不然就不予理睬。这样，最终就会找到目标设备。如前所述，这个过程仅用于设备的配置阶段，一旦完成了配置，CPU 就直接通过有关的总线地址访问目标设备了。

同样，我们把 1 型头部中的其他一些寄存器也放到阅读有关代码时再介绍。

还有一种 2 型头部是用于“PCI-CardBus 桥”的，CardBus 是笔记本电脑中使用的总线，我们在这里不感兴趣。

由于 PCI 总线的重要性和特殊性，PC 机制造商们在 BIOS 中提供了 PCI 总线操作，即 PCI 总线和设备的枚举以及配置所需的功能与服务。其中 PCI 总线和设备的枚举放在系统加电以后的自检阶段，而对配置过程所需的基本操作则以类似于 BIOS 调用的形式提供服务。考虑到现代的 PC 机操作系统都已采用保护模式和页式映射，BIOS 中还提供了一个特殊的 PCI 操作调用入口，供运行于保护模式和页式映射的 CPU 调用。提供了这些功能与服务的 BIOS 称为“PCI BIOS”。早期的 Linux 内核也依靠 BIOS 完成对 PCI 的枚举和配置操作，但是后来已实现了自己的 PCI 总线操作而不再依赖于 BIOS。现在，是否采用 PCI BIOS 提供的服务是个条件编译选择项。我们在下面将阅读 Linux 内核自己的 PCI 总线操作的代码，因为只有这样才能真正搞清有关的机理，否则一进入 BIOS 就进了黑盒子，不知道到底在干些什么了。

不过，正因为 Linux 内核以前也采用 PCI BIOS 提供的服务，所以有很多函数的函数名都带有前缀 pcibios。尽管现在已经可以不再涉及 BIOS，却还是保留了其中一些函数原来的函数名不变，以免引起混乱。

我们先看几个低层的函数，这些函数就是通过寄存器 0xCF8 和 0xCFC 读写目标设备上的配置寄存器时使用的。对配置寄存器的读写可以是按字节、按 16 位字、按 32 位长字的读写。具体的函数由 gcc 在编译时的预处理中根据宏定义生成，这个宏定义在 drivers/pci/pci.c 中：

```
478    /*
479     * Wrappers for all PCI configuration access functions.  They just check
480     * alignment, do locking and call the low-level functions pointed to
481     * by pci_dev->ops.
482     */
483
484    #define PCI_byte_BAD 0
485    #define PCI_word_BAD (pos & 1)
486    #define PCI_dword_BAD (pos & 3)
487
```

```
488    #define PCI_OP(rw, size, type) \
489    int pci_##rw##_config_##size (struct pci_dev *dev, int pos, type value) \
490    {                                        \
491        int res;                             \
492        unsigned long flags;                 \
493        if (PCI_##size##_BAD) return PCIBIOS_BAD_REGISTER_NUMBER;   \
494        spin_lock_irqsave(&pci_lock, flags);              \
495        res = dev->bus->ops->rw##_##size(dev, pos, value);      \
496        spin_unlock_irqrestore(&pci_lock, flags);         \
497        return res;                          \
498    }
499
500    PCI_OP(read, byte, u8 *)
501    PCI_OP(read, word, u16 *)
502    PCI_OP(read, dword, u32 *)
503    PCI_OP(write, byte, u8)
504    PCI_OP(write, word, u16)
```

以 501 行为例，经过 gcc 预处理的字符串替换就变成了函数 pci_read_config_word( )的定义：

```
int pci_read_config_word(struct pci_dev *dev, int pos, u16* value)
{
    int res;
    unsigned long flags;
    if (PCI_word_BAD) return PCIBIOS_BAD_REGISTER_NUMBER;
    spin_lock_irqsave(&pci_lock, flags);
    res = dev->bus->ops->read_word(dev, pos, value);
    spin_unlock_irqrestore(&pci_lock, flags);
    return res;
}
```

同样地，其他几行就变成了 pci_read_config_byte( )、pci_read_config_dword( )、pci_write_config_byte( )、pci_write_config_word( )等函数的定义。

再看 pci_read_config_word( )的代码。首先通过宏操作 PCI_word_BAD 检查地址 pos 是否与 16 位字边界对齐，然后就在加锁且不允许中断的条件下通过由具体设备提供的函数指针完成操作。这里参数 dev 指向代表着目标设备的 pci_dev 数据结构，读者在后面将会看到 pci_dev 数据结构的定义，这里我们先简单地介绍一下所涉及的几个字段。这个数据结构中有个指针 bus，指向代表着设备所在总线的 pci_bus 数据结构(见后)，而 pci_bus 结构中又有个指针 ops，指向一个 pci_ops 数据结构，这个数据结构的定义在 include/linux/pci.h 中：

```
424    /* Low-level architecture-dependent routines */
425
426    struct pci_ops {
427        int (*read_byte)(struct pci_dev *, int where, u8 *val);
428        int (*read_word)(struct pci_dev *, int where, u16 *val);
```

```
429        int (*read_dword)(struct pci_dev *, int where, u32 *val);
430        int (*write_byte)(struct pci_dev *, int where, u8 val);
431        int (*write_word)(struct pci_dev *, int where, u16 val);
432        int (*write dword)(struct pci_dev *, int where, u32 val);
433    };
```

显然，这是一个函数跳转表，与我们在前几章中看到过的一些数据结构相似。这个数据结构中提供的函数指针都是用来读/写 PCI 设备的配置寄存器的。内核中有三个 pci_ops 数据结构，分别用于所谓"1 型"和"2 型"的 PCI 配置寄存器操作（注意不是 1 型和 2 型的头部），以及通过 BIOS 完成的操作，均定义于 arch/i386/kernel/pci-pc.c。CPU 在初始化时根据条件编译或对硬件的测试从这三个数据结构中选择使用其一。在 PCI 总线发展的早期，有些"宿主—PCI 桥"曾经用过"2 型"PCI 配置寄存器操作，但是后来在 PCI 总线标准中已经明文规定不再生产此类"宿主—PCI 桥"。所以，pci_direct_conf2 只是为了与可能还在使用中的某些老式母板兼容而设的，实际上一般都会选择 pci_direct_conf1。

```
82     static struct pci_ops pci_direct_conf1 = {
83         pci_conf1_read_config_byte,
84         pci_conf1_read_config_word,
85         pci conf1 read_config_dword,
86         pci_conf1 write_config_byte,
87         pci_conf1_write config word,
88         pci_conf1_write config_dword
89     };
```

可见，上述的 pci_read_config_word( )是通过 pci_conf1_read_config_word( )完成的，其代码在 arch/i386/kernel/pci-pc.c 中：

```
45     static int pci_conf1_read_config_word(struct pci_dev *dev, int where, u16 *value)
46     {
47         outl(CONFIG_CMD(dev,where), 0xCF8);
48         *value = inw(0xCFC + (where&2));
49         return PCIBIOS_SUCCESSFUL;
50     }
```

这就是前面讲的通过 0xCF8 和 0xCFC 两个寄存器访问配置寄存器的过程。至于另两个数据结构，我们在这里不感兴趣，但还是列出于下供有需要的读者参考(均定义于 pci-pc.c)。

```
153    static struct pci_ops pci_direct_conf2 = {
154        pci_conf2 read config_byte,
155        pci_conf2_read_config word,
156        pci_conf2_read_config_dword,
157        pci_conf2_write_config_byte,
158        pci_conf2_write_config_word,
159        pci_conf2_write_config_dword
160    };
```

```
531     /*
532      * Function table for BIOS32 access
533      */
534
535     static struct pci_ops pci_bios_access = {
536             pci_bios_read_config_byte,
537             pci_bios_read_config_word,
538             pci_bios_read_config_dword,
539             pci_bios_write_config_byte,
540             pci_bios_write_config_word,
541             pci_bios_write_config_dword
542     };
```

可见，如果要由 BIOS 完成对 PCI 设备上配置寄存器的读写，则通过一个函数 pci_bios_read_config_word( )进入 BIOS，这个函数的代码也在同一文件(pci-pc.c)中：

```
441     static int pci_bios_read_config_word(struct pci_dev *dev, int where, u16 *value)
442     {
443         unsigned long ret;
444         unsigned long bx = (dev->bus->number << 8) | dev->devfn;
445
446         __asm__("lcall (%%esi); cld\n\t"
447             "jc 1f\n\t"
448             "xor %%ah, %%ah\n"
449             "1:"
450             : "=c" (*value),
451               "=a" (ret)
452             : "1" (PCIBIOS_READ_CONFIG_WORD),
453               "b" (bx),
454               "D" ((long) where),
455               "S" (&pci_indirect));
456         return (int) (ret & 0xff00) >> 8;
457     }
```

这段汇编代码通过一个指向目标地址和段地址的结构指针 pci_indirect 调用 BIOS 中的一个函数，这个函数就是 BIOS 为保护模式提供的 PCI 配置寄存器操作的总入口，其地址是通过扫描 PCI BIOS 的存储区间，根据若干特殊的字节(称为“签名”)而找到的。

除由 gcc 生成的上述这一组函数外，内核中还有另一组类似的、但是版本较老的函数 pcibios_read_config_byte( )、pcibios_read_config_word( )、pcibios_read_config_dword( )、pcibios_write_config_byte( )、pcibios_write_config_word( )、pcibios_write_config_dword( )，这是为了与老版本的动态安装模块兼容而保留的，定义于 drivers/pci/compat.c：

```
51      #define PCI_OP(rw, size, type)                                    \
52      int pcibios_##rw##_config_##size(unsigned char bus, unsigned char dev_fn, \
53                              unsigned char where, unsigned type val)   \
```

```
54    {                                               \
55        struct pci_dev *dev = pci_find_slot(bus, dev_fn);           \
56        if (!dev) return PCIBIOS_DEVICE_NOT_FOUND;              \
57        return pci_##rw##_config_##size(dev, where, val);           \
58    }
59
60    PCI_OP(read, byte, char *)
61    PCI_OP(read, word, short *)
62    PCI_OP(read, dword, int *)
63    PCI_OP(write, byte, char)
64    PCI_OP(write, word, short)
65    PCI_OP(write, dword, int)
```

以 61 行为例，经过预处理以后就变成了这样：

```
int pcibios_read_config_word (unsigned char bus, unsigned char dev fn,
                    unsigned char where, unsigned type val)
{
   struct pci_dev *dev = pci_find_slot(bus, dev_fn);
   if (!dev) return PCIBIOS_DEVICE_NOT_FOUND;
   return pci_read_config_word(dev, where, val);
}
```

可见，最后还是“九九归一”，落实到新版本的 pci_read_config_word( )上。

对 PCI 设备的枚举和配置都是在初始化阶段中完成的。一旦完成了初始化，以后的操作就比较简单了。PCI 总线的初始化由 pci_init( )完成，其代码在 drivers/pci/pci.c 中：

```
1162    void __init pci_init(void)
1163    {
1164        struct pci_dev *dev;
1165
1166        pcibios_init( );
1167
1168        pci_for_each_dev(dev) {
1169            pci_fixup_device(PCI_FIXUP_FINAL, dev);
1170        }
1171
1172    #ifdef CONFIG_PM
1173        pm_register(PM_PCI_DEV, 0, pci_pm_callback);
1174    #endif
1175    }
```

先提一下这里的条件编译选择项 CONFIG_PM，这是为电源管理而设的，PM 就是“Power Management”的意思，不过我们在这里对此不感兴趣。

一般而言，只要是采用 PCI 总线的 PC 机，其 BIOS 就必须提供对 PCI 总线操作的支持，因而称为

PCI BIOS。这种 BIOS 在机器加电以后的自检阶段会从系统中的第一个 PCI 桥，即“宿主—PCI 桥”开始进行探测和扫描，逐个地“枚举”(enumerate)连接在第一条 PCI 总线上的所有 PCI 设备并记录在案。如果其中的某个设备是个“PCI-PCI 桥”，则又前进一步，再探测和扫描连在这个桥上的次级 PCI 总线。就这样递归下去，直到穷尽系统中的所有 PCI 设备。其结果是在内存中建立起一棵代表着这些 PCI 总线和设备的“PCI 树”。一般的 PC 系统结构中都只有一个“宿主—PCI 桥”，但是如果有不止一个，也对其如法炮制。然后，操作系统可以通过 BIOS 调用来获取有关本系统中 PCI 设备的信息。当操作系统要进行对 PCI 设备的配置(config)操作时，也可以通过 BIOS 调用来完成。

但是，在实践中发现有些母板上的 PCI BIOS 有这样那样的问题，再说有些系统（如一些嵌入式系统）根本就没有 BIOS，所以后来 Linux 内核也提供了绕过 BIOS、直接探测和枚举 PCI 设备(以及配置)的功能，而把是否通过 BIOS 进行探测和枚举作为一个编译选择项。不过，尽管直接的 PCI 设备探测与枚举跟 BIOS 并无关系，代码的作者还是把这部分操作也放在 pcibios_init( )中完成，以求跟老的函数名一致。

函数 pcibios_init( )的代码在 arch/i386/kernel/pci-pc.c 中：

[pci_init( ) > pcibios_init( )]

```
953     /*
954      * Initialization. Try all known PCI access methods. Note that we support
955      * using both PCI BIOS and direct access: in such cases, we use I/O ports
956      * to access config space, but we still keep BIOS order of cards to be
957      * compatible with 2.0.X. This should go away some day.
958      */
959
960     void __init pcibios_init(void)
961     {
962         struct pci_ops *bios = NULL;
963         struct pci_ops *dir = NULL;
964
965     #ifdef CONFIG_PCI_BIOS
966         if ((pci_probe & PCI_PROBE_BIOS) && ((bios = pci_find_bios( )))) {
967             pci_probe |= PCI_BIOS_SORT;
968             pci_bios_present = 1;
969         }
970     #endif
971     #ifdef CONFIG_PCI_DIRECT
972         if (pci_probe & (PCI_PROBE_CONF1 | PCI_PROBE_CONF2))
973             dir = pci_check_direct( );
974     #endif
975         if (dir)
976             pci_root_ops = dir;
977         else if (bios)
978             pci_root_ops = bios;
979         else {
980             printk("PCI: No PCI bus detected\n");
981             return;
```

```
982         }
983 
984         printk("PCI: Probing PCI hardware\n");
985         pci_root_bus = pci_scan_bus(0, pci_root_ops, NULL);
986 
987         pcibios_irq_init( );
988         pcibios_fixup_peer_bridges( );
989         pcibios_fixup_irqs( );
990         pcibios_resource_survey( );
991 
992     #ifdef CONFIG_PCI_BIOS
993         if ((pci_probe & PCI_BIOS_SORT) && !(pci_probe & PCI_NO_SORT))
994             pcibios_sort( );
995     #endif
996     }
```

编译选择项 CONFIG_PCI_BIOS 表示通过 BIOS 进行 PCI 设备的探测和枚举，CONFIG_PCI_DIRECT表示直接进行PCI设备的探测和枚举。二者不是互斥的，也可以先试着通过BIOS探测，若不成功再亲自动手。我们对通过 PCI BIOS 进行的操作不感兴趣，但还是把 pci_find_bios( )的代码列在下面供读者参考，一方面也让读者对怎样发现 PCI BIOS 并找到 PCI 配置操作的入口有个感性的认识(arch/i386/kernel/pci-pc.c)。

[pci_init( ) > pcibios_init( ) > pci_find_bios( )]

```
544     /*
545      * Try to find PCI BIOS.
546      */
547 
548     static struct pci_ops * __init pci_find_bios(void)
549     {
550         union bios32 *check;
551         unsigned char sum;
552         int i, length;
553 
554         /*
555          * Follow the standard procedure for locating the BIOS32 Service
556          * directory by scanning the permissible address range from
557          * 0xe0000 through 0xfffff for a valid BIOS32 structure.
558          */
559 
560         for (check = (union bios32 *) __va(0xe0000);
561              check <= (union bios32 *) __va(0xffff0);
562              ++check) {
563             if (check->fields.signature != BIOS32_SIGNATURE)
564                 continue;
565             length = check->fields.length * 16;
566             if (!length)
```

```
567                 continue;
568             sum = 0;
569             for (i = 0; i < length ; ++i)
570                 sum += check->chars[i];
571             if (sum != 0)
572                 continue;
573             if (check->fields.revision != 0) {
574                 printk("PCI: unsupported BIOS32 revision %d at 0x%p, \
                                                report to <mj@suse.cz>\n",
575                     check->fields.revision, check);
576                 continue;
577             }
578             DBG("PCI: BIOS32 Service Directory structure at 0x%p\n", check);
579             if (check->fields.entry >= 0x100000) {
580                 printk("PCI: BIOS32 entry (0x%p) in high memory, cannot use.\n", check);
581                 return NULL;
582             } else {
583                 unsigned long bios32_entry = check->fields.entry;
584                 DBG("PCI: BIOS32 Service Directory entry at 0x%lx\n", bios32_entry);
585                 bios32_indirect.address = bios32_entry + PAGE_OFFSET;
586                 if (check_pcibios( ))
587                     return &pci_bios_access;
588             }
589             break;  /* Hopefully more than one BIOS32 cannot happen... */
590         }
591
592         return NULL;
593     }
```

这里 BIOS32_SIGNATURE 是一串特殊的字节，定义于同一文件(pci-pc.c)中：

```
259     /* BIOS32 signature: "_32_" */
260     #define BIOS32_SIGNATURE    \(('_' << 0) + ('3' << 8) + ('2' << 16) + ('_' << 24))
```

我们的重点是对 PCI 设备的直接探测与枚举，这是由 pci_check_direct( )完成的，其代码也在 pci-pc.c 中：

[pci_init( ) > pcibios_init( ) > pci_check_direct ( )]

```
192     static struct pci_ops * __init pci_check_direct(void)
193     {
194         unsigned int tmp;
195         unsigned long flags;
196
197         __save_flags(flags); __cli( );
198
199         /*
```

```
200          * Check if configuration type 1 works.
201          */
202         if (pci_probe & PCI_PROBE_CONF1) {
203             outb (0x01, 0xCFB);
204             tmp = inl (0xCF8);
205             outl (0x80000000, 0xCF8);
206             if (inl (0xCF8) == 0x80000000 &&
207                 pci_sanity_check(&pci_direct_conf1)) {
208                 outl (tmp, 0xCF8);
209                 __restore_flags(flags);
210                 printk("PCI: Using configuration type 1\n");
211                 request_region(0xCF8, 8, "PCI conf1");
212                 return &pci_direct_conf1;
213             }
214             outl (tmp, 0xCF8);
215         }
216 
217         /*
218          * Check if configuration type 2 works.
219          */
220         if (pci_probe & PCI_PROBE_CONF2) {
221             outb (0x00, 0xCFB);
222             outb (0x00, 0xCF8);
223             outb (0x00, 0xCFA);
224             if (inb (0xCF8) == 0x00 && inb (0xCFA) == 0x00 &&
225                 pci_sanity_check(&pci_direct_conf2)) {
226                 __restore_flags(flags);
227                 printk("PCI: Using configuration type 2\n");
228                 request_region(0xCF8, 4, "PCI conf2");
229                 return &pci_direct_conf2;
230             }
231         }
232 
233         __restore_flags(flags);
234         return NULL;
235     }
```

如前所述，“宿主—PCI 桥”的 I/O 口一定在 I/O 地址为 0xCF8-0xCFF 处，其中 0x8-0xCFB 为地址口，0xCFC-0CFF 为数据口。但是，也有可能系统中根本就没有 PCI 总线存在；或者再进一步，没有 PCI 总线存在，却凑巧有个 ISA 设备正在使用这些地址。针对这些可能性，PCI 总线标准规定了测试的方法，所以这里先按 1 型操作试试，如果成功就从 212 行返回了，不成功则再按 2 型试试。不过，如前所述，2 型操作现在已经不用了，所以对 1 型操作的测试一般总能成功。

探测到一个 1 型“宿主—PCI 桥”以后，要进一步采用 pci_direct_conf1 通过 pci_sanity_check( )再加验证。这个函数的代码也在 arch/i386/kernel/pci-pc.c 中：

[pci_init( ) > pcibios_init( ) > pci_check_direct ( ) > pci_sanity_check( )]

```
162    /*
163     * Before we decide to use direct hardware access mechanisms, we try to do some
164     * trivial checks to ensure it at least _seems_ to be working - we just test
165     * whether bus 00 contains a host bridge (this is similar to checking
166     * techniques used in XFree86, but ours should be more reliable since we
167     * attempt to make use of direct access hints provided by the PCI BIOS).
168     *
169     * This should be close to trivial, but it isn't, because there are buggy
170     * chipsets(yes, you guessed it, by Intel and Compaq)that have no class ID.
171     */
172    static int __init pci_sanity_check(struct pci_ops *o)
173    {
174        u16 x;
175        struct pci_bus bus;      /* Fake bus and device */
176        struct pci_dev dev;
177
178        if (pci_probe & PCI_NO_CHECKS)
179            return 1;
180        bus.number = 0;
181        dev.bus = &bus;
182        for(dev.devfn=0; dev.devfn < 0x100; dev.devfn++)
183            if ((!o->read_word(&dev, PCI_CLASS_DEVICE, &x) &&
184                 (x == PCI_CLASS_BRIDGE_HOST || x == PCI_CLASS_DISPLAY_VGA)) ||
185                (!o->read_word(&dev, PCI_VENDOR_ID, &x) &&
186                 (x == PCI_VENDOR_ID_INTEL || x == PCI_VENDOR_ID_COMPAQ)))
187                return 1;
188        DBG("PCI: Sanity check failed\n");
189        return 0;
190    }
```

怎样验证呢？我们知道，一条 PCI 总线上最多可以有 256 种功能，或者说 256 个逻辑设备，而“宿主—PCI 桥”至少应该提供 PCI_CLASS_BRIDGE_HOST 和 PCI_CLASS_DISPLAY_VGA 这二者之一。如果都没有，那么至少这“宿主—PCI 桥”芯片应该是 Intel 或 Compaq 制造的。如果连这也得不到证实，那就还是只能认为“宿主—PCI 桥”实际上并不存在，前面从 PCI 口中读到的信息只是碰巧而已。

“宿主—PCI 桥”的存在得到验证以后，系统中就多了一项 I/O 设备，占据的 I/O 地址区间从 0xCF8 开始，长度为 8。所以，这里通过 request_region( )在内核的 I/O 设备资源树(见后)中增加一个节点。表示这个 I/O 地址区间已由“PCI conf1”占用。

如果成功地发现了一个 1 型“宿主—PCI 桥”，函数最终返回指向 pci_direct_conf1 的指针。这个 pci_ops 数据结构的定义已经在前面看到过了。从此以后，在枚举和配置操作中采用什么方法访问 PCI 总线也就定下来了。

回到 pcibios_init( )的代码中，接着(985 行)就是对 PCI 总线的扫描，即对连接在这条总线上的 PCI 设备的探测与枚举了，这是由 pci_scan_bus( )完成的，其代码见 drivers/pci/pci.c：

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( )]

```
1045    struct pci_bus * __init pci_scan_bus(int bus, struct pci_ops *ops, void *sysdata)
1046    {
1047        struct pci_bus *b = pci_alloc_primary_bus(bus);
1048        if (b) {
1049            b->sysdata = sysdata;
1050            b->ops = ops;
1051            b->subordinate = pci_do_scan_bus(b);
1052        }
1053        return b;
1054    }
```

“宿主—PCI 桥”后面是系统中的第一条 PCI 总线，称为“主 PCI 总线”。在内核中，每条 PCI 总线都由一个 pci_bus 数据结构代表，所以先要为之分配一个数据结构，这种数据结构定义于 include/linux/pci.h 中：

```
381     struct pci_bus {
382         struct list_head node;      /* node in list of buses */
383         struct pci_bus  *parent;    /* parent bus this bridge is on */
384         struct list_head children;  /* list of child buses */
385         struct list_head devices;   /* list of devices on this bus */
386         struct pci_dev  *self;      /* bridge device as seen by parent */
387         struct resource *resource[4];   /* address space routed to this bus */
388
389         struct pci_ops  *ops;       /* configuration access functions */
390         void            *sysdata;   /* hook for sys-specific extension */
391         struct proc_dir_entry *procdir; /* directory entry in /proc/bus/pci */
392
393         unsigned char   number;     /* bus number */
394         unsigned char   primary;    /* number of primary bridge */
395         unsigned char   secondary;  /* number of secondary bridge */
396         unsigned char   subordinate;    /* max number of subordinate buses */
397
398         char            name[48];
399         unsigned short  vendor;
400         unsigned short  device;
401         unsigned int    serial;     /* serial number */
402         unsigned char   pnpver;     /* Plug & Play version */
403         unsigned char   productver; /* product version */
404         unsigned char   checksum;   /* if zero - checksum passed */
405         unsigned char   pad1;
406     };
```

系统中的每条 PCI 总线都有个编号 number，主 PCI 总线的编号为 0。

所有的 pci_bus 数据结构都互相连接在一起，形成若干(通常只有一棵)PCI 总线树，每棵树的根是一个代表着“宿主—PCI 桥”的 pci_bus 结构。内核中有一个队列头 pci_root_buses，所有代表着“宿主—PCI 桥”的 pci_bus 结构都通过其内部的队列头 node 挂在这个队列中。同时，每个 pci_bus 结构本身

又维持着两个队列。一个是 devices，凡是连接在这条总线上的设备都有个 pci_dev 数据结构(见下)挂在这个队列中。另一个是 children，凡是通过“PCI-PCI 桥” 连接在这条总线上的次层 PCI 总线都有个 pci_bus 数据结构挂在这个队列中。这样，从队列 pci_root_buses 开始的整个层次结构就反映着系统中 PCI 总线和设备的配备和连接。而 pci_scan_bus( )的目的正是要在内存中建立起这样一个层次结构。通常系统中只有一个“宿主—PCI 桥”，所以 pci_root_buses 中通常只有一个节点，也就是只有一棵树。

每个 PCI 设备都由一个 pci_dev 数据结构代表，这种数据结构定义于 include\linux\pci.h：

```
311	/*
312	 * The pci_dev structure is used to describe both PCI and ISAPnP devices.
313	 */
314	struct pci_dev {
315		struct list_head global_list;	/* node in list of all PCI devices */
316		struct list_head bus_list;	/* node in per-bus list */
317		struct pci_bus	*bus;		/* bus this device is on */
318		struct pci_bus	*subordinate;	/* bus this device bridges to */
319	
320		void		*sysdata;	/* hook for sys-specific extension */
321		struct proc_dir_entry *procent; /* device entry in /proc/bus/pci */
322	
323		unsigned int	devfn;		/* encoded device & function index */
324		unsigned short	vendor;
325		unsigned short	device;
326		unsigned short	subsystem_vendor;
327		unsigned short	subsystem_device;
328		unsigned int	class;		/* 3 bytes: (base,sub,prog-if) */
329		u8		hdr_type;	/* PCI header type (`multi' flag masked out) */
330		u8		rom_base_reg;	/* which config register controls the ROM */
331	
332		struct pci_driver  *driver;/* which driver has allocated this device*/
333		void		*driver_data;	/* data private to the driver */
334		dma_addr_t	dma_mask;	/* Mask of the bits of bus address this
335					   device implements.  Normally this is
336					   0xffffffff.  You only need to change
337					   this if your device has broken DMA
338					   or supports 64-bit transfers.  */
339	
340		/* device is compatible with these IDs */
341		unsigned short vendor_compatible[DEVICE_COUNT_COMPATIBLE];
342		unsigned short device_compatible[DEVICE_COUNT_COMPATIBLE];
343	
344		/*
345		 * Instead of touching interrupt line and base address registers
346		 * directly, use the values stored here. They might be different!
347		 */
348		unsigned int	irq;
349		struct resource resource[DEVICE_COUNT_RESOURCE];
```

```
                                    /* I/O and memory regions + expansion ROMs */
350         struct resource dma_resource[DEVICE_COUNT_DMA];
351         struct resource irq_resource[DEVICE_COUNT_IRQ];
352
353         char            name[80];       /* device name */
354         char            slot_name[8];   /* slot name */
355         int             active;         /* ISAPnP: device is active */
356         int             ro;             /* ISAPnP: read only */
357         unsigned short  regs;           /* ISAPnP: supported registers */
358
359         int  (*prepare)(struct pci_dev *dev);    /* ISAPnP hooks */
360         int  (*activate)(struct pci_dev *dev);
361         int  (*deactivate)(struct pci_dev *dev);
362     };
```

每个 pci_dev 数据结构都同时连入两个队列，一方面通过 global_list 挂入一个总的 pci_dev 结构队列；同时又通过 bus_list 挂入其所在总线的 pci_dev 结构队列 devices，并且使指针 bus 指向代表着该总线的 pci_bus 数据结构。如果具体的设备是个“PCI-PCI 桥”则还要使其指针 subordinate 指向代表着另一个 PCI 总线(次层 PCI 总线)的 pci_bus 数据结构。

在 pci_bus 结构中有个 resource 数据结构数组，而 pci_dev 结构中更有三个这样的数组。每个 resource 数据结构都可以用来描述一个地址区间，包括其起点和终点。地址本身就是一种重要的“资源”；对于 PCI 总线的初始化，目的就在于为各个设备上的各个区间分配和设置地址，以代替从前手工设置跳线或小开关的过程，因而在这个过程中所涉及的首要资源就是地址，这种数据结构即因此而得名。从某种意义上说，PCI 总线的整个初始化过程就是与这些地址区间打交道。关于 resource 数据结构后面还要详细介绍。

随着代码的阅读，读者自会明白上面这两个数据结构中其他一些字段的作用。

函数 pci_alloc_primary_bus( )的代码在 drivers/pci/pci.c 中：

```
1026    struct pci_bus * __init pci_alloc_primary_bus(int bus)
1027    {
1028        struct pci_bus *b;
1029
1030        if (pci_bus_exists(&pci_root_buses, bus)) {
1031            /* If we already got to this bus through a different bridge, ignore it */
1032            DBG("PCI: Bus %02x already known\n", bus);
1033            return NULL;
1034        }
1035
1036        b = pci_alloc_bus( );
1037        list_add_tail(&b->node, &pci_root_buses);
1038
1039        b->number = b->secondary = bus;
1040        b->resource[0] = &ioport_resource;
1041        b->resource[1] = &iomem_resource;
1042        return b;
```

```
1043    }
```

同一 PCI 总线只能由一个 pci_bus 结构代表，在 PCI 树中或 pci_root_buses 队列中只能出现一次，所以先要通过 pci_bus_exists( )检查是否重复。参数 bus 为相应 PCI 总线的编号，在这里是 0（见前面的 985 行）。这里引用的 ioport_resource 和 iomem_resource 分别为反映着 I/O 空间和内存空间地址占用情况的两棵资源树，后面我们还要讲到。

为主 PCI 总线分配了数据结构以后，就可以开始扫描了。函数 pci_do_scan_bus( )的代码在 drivers/pci/pci.c 中：

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( ) > pci_do_scan_bus( )]

```
970     static unsigned int __init pci_do_scan_bus(struct pci_bus *bus)
971     {
972         unsigned int devfn, max, pass;
973         struct list_head *ln;
974         struct pci_dev *dev, dev0;
975
976         DBG("Scanning bus %02x\n", bus->number);
977         max = bus->secondary;
978
979         /* Create a device template */
980         memset(&dev0, 0, sizeof(dev0));
981         dev0.bus = bus;
982         dev0.sysdata = bus->sysdata;
983
984         /* Go find them, Rover! */
985         for (devfn = 0; devfn < 0x100; devfn += 8) {
986             dev0.devfn = devfn;
987             pci_scan_slot(&dev0);
988         }
989
990         /*
991          * After performing arch-dependent fixup of the bus, look behind
992          * all PCI-to-PCI bridges on this bus.
993          */
994         DBG("Fixups for bus %02x\n", bus->number);
995         pcibios_fixup_bus(bus);
996         for (pass=0; pass < 2; pass++)
997             for (ln=bus->devices.next; ln != &bus->devices; ln=ln->next) {
998                 dev = pci_dev_b(ln);
999                 if (dev->hdr_type == PCI_HEADER_TYPE_BRIDGE ||
                                        dev->hdr_type == PCI_HEADER_TYPE_CARDBUS)
1000                    max = pci_scan_bridge(bus, dev, max, pass);
1001            }
1002
1003        /*
```

```
1004         * We've scanned the bus and so we know all about what's on
1005         * the other side of any bridges that may be on this bus plus
1006         * any devices.
1007         *
1008         * Return how far we've got finding sub-buses.
1009         */
1010        DBG("Bus scan for %02x returning with max=%02x\n", bus->number, max);
1011        return max;
1012    }
```

扫描一条 PCI 总线的直接目的就是逐个地发现连接在该总线上的 PCI 设备，为其建立起 pci_dev 数据结构并挂入相应的队列，这就是所谓枚举。所以，这里先准备下一个空白的 pci_dev 结构，然后依次对各个 PCI 接口通过 pci_scan_slot( )扫描，每次扫描 8 个功能，即 8 个逻辑设备，这是每块 PCI 接口卡上的最大容量。注意这里的 devfn 是将前述的 5 位设备号与 3 位功能号合在一起的“逻辑设备号”。这个函数的定义也在 drivers/pci/pci.c 中：

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( ) > pci_do_scan_bus( ) > pci_scan_slot( )]

```
932     struct pci_dev * __init pci_scan_slot(struct pci_dev *temp)
933     {
934         struct pci_bus *bus = temp->bus;
935         struct pci_dev *dev;
936         struct pci_dev *first_dev = NULL;
937         int func = 0;
938         int is_multi = 0;
939         u8 hdr_type;
940 
941         for (func = 0; func < 8; func++, temp->devfn++) {
942             if (func && !is_multi)      /* not a multi-function device */
943                 continue;
944             if (pci_read_config_byte(temp, PCI_HEADER_TYPE, &hdr_type))
945                 continue;
946             temp->hdr_type = hdr_type & 0x7f;
947 
948             dev = pci_scan_device(temp);
949             if (!dev)
950                 continue;
951             pci_name_device(dev);
952             if (!func) {
953                 is_multi = hdr_type & 0x80;
954                 first_dev = dev;
955             }
956 
957             /*
958              * Link the device to both the global PCI device chain and
959              * the per-bus list of devices.
960              */
```

```
961             list_add_tail(&dev->global_list, &pci_devices);
962             list_add_tail(&dev->bus_list, &bus->devices);
963
964             /* Fix up broken headers */
965             pci_fixup_device(PCI_FIXUP_HEADER, dev);
966         }
967         return first_dev;
968     }
```

一个物理 PCI 设备(接口卡)可以是多功能的，也可以是单功能的，如果是单功能的则其逻辑设备号必是 8 的倍数(因为低 3 位的功能号为 0)，而一个设备是否为多功能要从设备读入其头部类型以后才能知道(头部类型字节的最高位为 1 表示多功能)，所以先假定为单功能。然后，就以 pci_bus 结构指针 temp 为参数通过 pci_read_config_byte( )读设备的头部字节，注意这里的 temp->devfn 为目标逻辑设备号。如前所述，这个函数最后会通过由一个 pci_ops 结构提供的函数指针完成操作。读到了头部类型字节以后，其低 7 位为类型编码(见 946 行)，接着就可以通过 pci_scan_device( )进一步读取具体逻辑设备的配置信息了(drivers/pci/pci.c)。前面讲过，每项逻辑设备的配置寄存器组中有些信息是由厂商提供、固化在里面的，有些信息(如地址映射和总线号)则有待设置。

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( ) > pci_do_scan_bus( ) > pci_scan_slot( ) > pci_scan_device( )]

```
898     /*
899      * Read the config data for a PCI device, sanity-check it
900      * and fill in the dev structure...
901      */
902     static struct pci_dev * __init pci_scan_device(struct pci_dev *temp)
903     {
904         struct pci_dev *dev;
905         u32 l;
906
907         if (pci_read_config_dword(temp, PCI_VENDOR_ID, &l))
908             return NULL;
909
910         /* some broken boards return 0 or ~0 if a slot is empty: */
911         if (l == 0xffffffff || l == 0x00000000 || l == 0x0000ffff || l == 0xffff0000)
912             return NULL;
913
914         dev = kmalloc(sizeof(*dev), GFP_KERNEL);
915         if (!dev)
916             return NULL;
917
918         memcpy(dev, temp, sizeof(*dev));
919         dev->vendor = l & 0xffff;
920         dev->device = (l >> 16) & 0xffff;
921
922         /* Assume 32-bit PCI; let 64-bit PCI cards (which are far rarer)
```

```
923             set this higher, assuming the system even supports it.  */
924         dev->dma_mask = 0xffffffff;
925         if (pci_setup_device(dev) < 0) {
926             kfree(dev);
927             dev = NULL;
928         }
929         return dev;
930     }
```

首先读入第一个长字，其低 16 位为厂商编号，高 16 位为设备号。只要这些编号不是全 1 或全 0，就可以认为是有效的编号，从而设备存在。所以要分配一个新的 pci_dev 数据结构并设置有关的字段，然后，通过 pci_setup_device( )进一步从 PCI 口读入有关这个设备的信息，并继续设置这个数据结构(drivers/pci/pci.c)。

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( ) > pci_do_scan_bus( ) > pci_scan_slot( ) > pci_scan_device( ) > pci_setup_device( )]

```
841     /*
842      * Fill in class and map information of a device
843      */
844     int pci_setup_device(struct pci_dev * dev)
845     {
846         u32 class;
847
848         sprintf(dev->slot_name, "%02x:%02x.%d", dev->bus->number,
                                PCI_SLOT(dev->devfn), PCI_FUNC(dev->devfn));
849         sprintf(dev->name, "PCI device %04x:%04x", dev->vendor, dev->device);
850
851         pci_read_config_dword(dev, PCI_CLASS_REVISION, &class);
852         class >>= 8;                    /* upper 3 bytes */
853         dev->class = class;
854         class >>= 8;
855
856         DBG("Found %02x:%02x [%04x/%04x] %06x %02x\n", dev->bus->number,
                   dev->devfn, dev->vendor, dev->device, class, dev->hdr_type);
857
858         switch (dev->hdr_type) {            /* header type */
859         case PCI_HEADER_TYPE_NORMAL:            /* standard header */
860             if (class == PCI_CLASS_BRIDGE_PCI)
861                 goto bad;
862             pci_read_irq(dev);
863             pci_read_bases(dev, 6, PCI_ROM_ADDRESS);
864             pci_read_config_word(dev, PCI_SUBSYSTEM_VENDOR_ID,
                                                &dev->subsystem_vendor);
865             pci_read_config_word(dev, PCI_SUBSYSTEM_ID, &dev->subsystem_device);
866             break;
867
```

```
868         case PCI_HEADER_TYPE_BRIDGE:            /* bridge header */
869             if (class != PCI_CLASS_BRIDGE_PCI)
870                 goto bad;
871             pci_read_bases(dev, 2, PCI_ROM_ADDRESS1);
872             break;
873
874         case PCI_HEADER_TYPE_CARDBUS:           /* CardBus bridge header */
875             if (class != PCI_CLASS_BRIDGE_CARDBUS)
876                 goto bad;
877             pci_read_irq(dev);
878             pci_read_bases(dev, 1, 0);
879             pci_read_config_word(dev, PCI_CB_SUBSYSTEM_VENDOR_ID,
                                                    &dev->subsystem_vendor);
880             pci_read_config_word(dev, PCI_CB_SUBSYSTEM_ID, &dev->subsystem_device);
881             break;
882
883         default:                        /* unknown header */
884             printk(KERN_ERR "PCI: device %s has unknown header type %02x, ignoring.\n",
885                 dev->slot_name, dev->hdr_type);
886             return -1;
887
888         bad:
889             printk(KERN_ERR
            "PCI: %s: class %x doesn't match header type %02x. Ignoring class.\n",
890                    dev->slot_name, class, dev->hdr_type);
891             dev->class = PCI_CLASS_NOT_DEFINED;
892         }
893
894         /* We found a fine healthy device, go go go... */
895         return 0;
896     }
```

接着是读入用于设备类别和版本号的长字。再往下就取决于具体设备的头部类型了(858 行)。头部类型 PCI_HEADER_TYPE_NORMAL 表示该设备为一般的 PCI 设备，PCI_HEADER_TYPE_BRIDGE 表示“PCI-PCI 桥”，PCI_HEADER_TYPE_CARDBUS 则表示“PCI-CardBus 桥”。

先看一般的 PCI 设备。PCI 设备通常都是可以发出中断请求的，所以设备配置寄存器组中有两个字节，即 PCI_INTERRUPT_PIN 和 PCI_INTERRUPT_LINE，反映着该设备的中断请求信号线与总线和系统的连接方式。在 PCI 总线上有 INTA～INTD 共 4 条中断请求线，从而在 PCI 插槽中有 4 根“针”。并非所有的 PCI 设备都能产生中断请求，例如图形卡就多半不能产生中断请求。如果 PCI_INTERRUPT_PIN 字节为 0，就表示本设备不能产生中断请求。但是，如果一个 PCI 设备能产生中断请求，那么在设备内部必定已经把中断请求连到 PCI 总线的某条中断请求线上，此时 PCI_INTERRUPT_PIN 字节的数值(1～4)表示该设备的中断请求连在哪一条线上。这种连接是由硬件决定的，所以 PCI_INTERRUPT_PIN 字节是个只读的字节，不能通过软件设置。可是，连在哪一条 PCI 中断请求线上只是事情的一个方面，还有个最终是连接到系统的中断控制器(8259A 或 APIC)上的哪一条中断请求线的问题，称为“中断请求路径”，这就是以前需要通过跳线或小开关设置的两个内容之一。

这是由软件选择和设置的，选择的结果就存储在 PCI_INTERRUPT_LINE 字节中。这里要指出，这个寄存器的目的只是保存信息，而并不带有控制功能。所以如果把这个寄存器的内容从 8 改成 9 并不意味着改变了连接的目标。

这里通过 pci_read_irq( )读入这两个字节，并把中断请求线号记录在 pci_dev 结构中(drivers/pci/pci.c)。

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( ) > pci_do_scan_bus( ) > pci_scan_slot( )
 > pci_scan_device( ) > pci_setup_device( ) > pci_read_irq( )]

```
827    /*
828     * Read interrupt line and base address registers.
829     * The architecture-dependent code can tweak these, of course.
830     */
831    static void pci_read_irq(struct pci_dev *dev)
832    {
833        unsigned char irq;
834
835        pci_read_config_byte(dev, PCI_INTERRUPT_PIN, &irq);
836        if (irq)
837            pci_read_config_byte(dev, PCI_INTERRUPT_LINE, &irq);
838        dev->irq = irq;
839    }
```

PCI 设备(接口)中一般都带有一些 RAM 和 ROM 区间，通常的控制/状态寄存器和数据寄存器也往往以 RAM 区间的形式出现，这些地址以后都要先映射到系统总线上，再进一步映射到内核的虚存空间。现在先要通过 pci_read_bases( )把这些区间的大小和当前的地址读进来。对于一般的 PCI 设备，最多可以有 6 个这样的 RAM 区间。函数 pci_read_bases( )的代码在 drivers/pci/pci.c 中，我们分两段阅读。

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( ) > pci_do_scan_bus( ) > pci_scan_slot( )
 > pci_scan_device( ) > pci_setup_device( ) > pci_read_bases( )]

```
547    static void pci_read_bases(struct pci_dev *dev, unsigned int howmany, int rom)
548    {
549        unsigned int pos, reg, next;
550        u32 l, sz;
551        struct resource *res;
552
553        for(pos=0; pos<howmany; pos = next) {
554            next = pos+1;
555            res = &dev->resource[pos];
556            res->name = dev->name;
557            reg = PCI_BASE_ADDRESS_0 + (pos << 2);
558            pci_read_config_dword(dev, reg, &l);
559            pci_write_config_dword(dev, reg, ~0);
560            pci_read_config_dword(dev, reg, &sz);
561            pci_write_config_dword(dev, reg, l);
```

```
562             if (!sz || sz == 0xffffffff)
563                 continue;
564             if (l == 0xffffffff)
565                 l = 0;
566             if((l & PCI_BASE_ADDRESS_SPACE)==PCI_BASE_ADDRESS_SPACE_MEMORY) {
567                 res->start = l & PCI_BASE_ADDRESS_MEM_MASK;
568                 sz = pci_size(sz, PCI_BASE_ADDRESS_MEM_MASK);
569             } else {
570                 res->start = l & PCI_BASE_ADDRESS_IO_MASK;
571                 sz = pci_size(sz, PCI_BASE_ADDRESS_IO_MASK & 0xffff);
572             }
573             res->end = res->start + (unsigned long) sz;
574             res->flags |= (l & 0xf) | pci_calc_resource_flags(l);
575             if ((l & (PCI_BASE_ADDRESS_SPACE | PCI_BASE_ADDRESS_MEM_TYPE_MASK))
576             == (PCI_BASE_ADDRESS_SPACE_MEMORY | PCI_BASE_ADDRESS_MEM_TYPE_64)) {
577                 pci_read_config_dword(dev, reg+4, &l);
578                 next++;
579     #if BITS_PER_LONG == 64
        . . . . . . .
588     #else
589                 if (l) {
590                     printk(KERN_ERR
    "PCI: Unable to handle 64-bit address for device %s\n", dev->slot_name);
591                     res->start = 0;
592                     res->flags = 0;
593                     continue;
594                 }
595     #endif
596             }
597         }
```

这里的 dev->resource[ ]是 pci_dev 数据结构中的一个 resource 结构数组，用来记录设备上各个地址区间的起始地址与长度。数组的大小为 12，这是因为除 PCI 设备的 6 个常规地址区间以外还可以有一个扩充 ROM 区间，同时还要考虑到设备为 PCI 桥时的需要。

在设备的配置寄存器组中，用于 6 个常规地址区间的长字都是连续的，所以可以通过一个 for 循环来读出，代码中的 557 行计算出各个长字的起始地址。操作时，先从相应的长字中读(558 行)，读得的数值是区间的(本地)起始地址与其他一些信息的组合，对这个数值的解释是：

- 首先，其最低位，即 bit0 表示区间的类型，为 0 表示是个存储器区间，为 1 则表示是个 I/O 地址(即寄存器)区间。注意这只是说可以通过什么样的操作(I/O 指令或访内指令)来访问这个区间，而与其内容是否为寄存器无关，寄存器也可以通过存储单元的形式来实现（称为“memory mapped”）。
- 如果是存储器区间，那么其高 28 位就是起始地址的高 28 位(起始地址的最低 4 位一定是 0)。
- 如果是 I/O 区间，那么其高 29 位就是起始地址的高 29 位(起始地址的最低 3 位一定是 0)。
- 对于存储器区间，如果 bit3 为 1，就表示对这个区间的操作可以流水线化，称为“可预取”

(prefetchable)，否则就不能流水线化，而只能一个单元一个单元地读写(见后)。

- 还有，bit2 为 1 表示采用 64 位地址，为 0 表示采用 32 位地址。
- 最后，bit1 为 1 表示区间的大小超过 1MB，为 0 则表示区间的大小在 1MB 以下。

下列的一些常数定义反映了对这些标志位的约定(include/linux/pci.h)：

```
84    #define  PCI_BASE_ADDRESS_SPACE          0x01     /* 0 = memory, 1 = I/O */
85    #define  PCI_BASE_ADDRESS_SPACE_IO       0x01
86    #define  PCI_BASE_ADDRESS_SPACE_MEMORY   0x00
87    #define  PCI_BASE_ADDRESS_MEM_TYPE_MASK  0x06
88    #define  PCI_BASE_ADDRESS_MEM_TYPE_32    0x00     /* 32 bit address */
89    #define  PCI_BASE_ADDRESS_MEM_TYPE_1M    0x02     /* Below 1M [obsolete] */
90    #define  PCI_BASE_ADDRESS_MEM_TYPE_64    0x04     /* 64 bit address */
91    #define  PCI_BASE_ADDRESS_MEM_PREFETCH   0x08     /* prefetchable? */
```

从配置寄存器组的一个长字中读出了区间的起始地址以后，往同一长字中写入全 1(559 行)，即 0xffffffff，接着再从同一长字中读(560 行)，这时候读得的数值便是区间的大小。这个数值的低 4 位或低 3 位为控制信息，这一点上与起始地址的格式相似，但是在其高 28 位或 29 位中只有位置最低的那位 1 才有效。区间的大小必定是 2 的某次幂，所以其二进制数值中应该只有一位是 1，而其他各位均为 0。但是，在读得的数值中却通常有多位是 1，此时只有位置最低的那个 1 才有效，所以要通过 pci_size( ) 加以换算(drivers/pci/pci.c)。

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( ) > pci_do_scan_bus( ) > pci_scan_slot( ) > pci_scan_device( ) > pci_setup_device( ) > pci_read_bases( ) > pci_size( )]

```
537    /*
538     * Find the extent of a PCI decode..
539     */
540    static u32 pci_size(u32 base, unsigned long mask)
541    {
542        u32 size = mask & base;      /* Find the significant bits */
543        size = size & ~(size-1);     /* Get the lowest of them to
                                           find the decode size */
544        return size-1;            /* extent = size - 1 */
545    }
```

这里先用 mask 把最低的 4 位或 3 位屏蔽掉，然后再把位置最低的那个 1 抽取出来。例如，假定把最低的 4 位或 3 位屏蔽掉以后得到 size 的数值是 0xffff0100，则(size－1)为 0xffff00ff，而~(size－1)就是 0x0000ff00，最后得到 size = 0xffff0100 & 0x0000ff00 = 0x100。这样，就把 size 的二进制数值中位置最低的那个 1 抽取了出来，从而得到 size 的实际数值为 0x100，即区间的大小为 256，而返回的值则为 255。读者一定会问，为什么不直接就读回 0x100，而要搞得这么复杂呢？当然，前面那些为 1 的位是有意义的。事实上，除位置最低的那个 1 表示着区间的大小以外，所有为 1 的位(包括位置最低的 1)都是可以(在建立地址映射时)加以设置的，而所有为 0 的位则不能设置。在这个例子中，就表示该区间在映射后的起始地址必须是 64KB 边界上的第一个或第二个 256 字节。我们在这里为了说明问题而举了

一个比较特殊的例子，多数情况下从位置最低的 1 开始往上所有各位都是 1，表示区间的起点是与区间的大小“自然”对齐的。

读取了区间大小以后，还要把起始地址写回这个长字(561 行)，恢复其原状。

然后，把从寄存器读出的内容换算成区间的起点和终点，记录在相应的 resource 结构中。

对“可预取”这个概念可能还需要一些说明。如上所述，一个区间之为“存储器区间”或“I/O 区间”取决于该区间的地址是在“存储器地址空间”或“I/O 地址空间”，实质上取决于 CPU 通过哪一类的指令访问这个区间，而与在具体地址上的是存储单元或寄存器无关。事实上，有些 CPU 根本就没有 I/O 指令，因而就没有 I/O 地址空间，所有的寄存器都在存储器地址空间中。另一方面，以前也讲过，i386 处理器虽有 I/O 地址空间，但是其 I/O 地址空间比较小(16 位)，因而比较拥挤。再说通过 I/O 指令访问就意味着每次访问时都得调用一个汇编语言子程序(因为 C 语言中没有相应的语言成分)，而不像通过访内指令时那样可以把寄存器看作一个变量，因而不那么方便，效率也要低一些。所以，在 PCI 设备中一般都倾向于将寄存器映射到存储器地址空间，而避免使用 I/O 地址空间。然而，即使都使用存储器地址空间，寄存器与真正的存储单元还是有着本质的不同，主要在于从一个单元读出时是否可能改变其内容。对于普通的存储单元，读操作是不会改变其内容的，所以反复从同一单元读出多次也没有关系，每次读出的内容都一样。而寄存器就不同了。首先，有些寄存器可能代表着一个 FIFO 队列，从中读出时每次都是读出 FIFO 中最前面的内容，因而每次都可能不同。其次，很多状态寄存器中的状态位在读出时就清成了 0。不过，二者间的这种区别对于个别的读操作，即每次只从一个单元的读出并无影响。但是，对于连续的、成块的读出就有影响了。为了提高成块读出的效率，PCI 总线对成块的读出加以流水线化，在应 CPU 的要求读出单元 N 后预先就把 N+1 也读入流水线，这样当 CPU 真的接着要求读出单元 N+1 时就不用等待了，这就是所谓“预取”。显然，预取是建立在对 CPU 意图的猜测上的，这种猜测有可能(通常)成真，也有可能失败。如果猜测失败了，预取的内容就丢掉了，对于存储单元这并无害处，下次需要时再读就是了。可是对寄存器就不同了，下次再读时很可能其内容已经因为预取而变化了。所以，一般而言，映射在存储器地址空间中的寄存器是不可预取的。

我们在这里不关心 64 位 PCI 总线设备，即采用 64 位地址的设备，所以跳过对此的检查和处理(575～596 行)。

除常规的 6 个存储区间外，设备上还可能提供一个扩充 ROM 区间，用于这个区间的长字与前面这 6 个不连在一起，所以要放在 for 循环外面单独处理。不过，所作的处理与前面并无大的不同，我们把下面这段代码留给读者。

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( ) > pci_do_scan_bus( ) > pci_scan_slot( ) > pci_scan_device( ) > pci_setup_device( ) > pci_read_bases( )]

```
598         if (rom) {
599             dev->rom_base_reg = rom;
600             res = &dev->resource[PCI_ROM_RESOURCE];
601             pci_read_config_dword(dev, rom, &l);
602             pci_write_config_dword(dev, rom, ~PCI_ROM_ADDRESS_ENABLE);
603             pci_read_config_dword(dev, rom, &sz);
604             pci_write_config_dword(dev, rom, l);
605             if (l == 0xffffffff)
606                 l = 0;
```

```
607             if (sz && sz != 0xffffffff) {
608                 res->flags = (l & PCI_ROM_ADDRESS_ENABLE) |
609                   IORESOURCE_MEM | IORESOURCE_PREFETCH |
                      IORESOURCE_READONLY | IORESOURCE_CACHEABLE;
610                 res->start = l & PCI_ROM_ADDRESS_MASK;
611                 sz = pci_size(sz, PCI_ROM_ADDRESS_MASK);
612                 res->end = res->start + (unsigned long) sz;
613             }
614             res->name = dev->name;
615         }
616     }
```

对 ROM 区间地址的解读与 RAM 区间的有所不同，其最低的 11 位都是保留的，其中 bit0 为 PCI_ROM_ADDRESS_ENABLE，为 1 时表示区间有效。

回到 pci_setup_device( )的代码中（864 行），还要把设备的子系统号和子系统厂商号也读进来。

这里对“PCI-PCI 桥”的处理(869~872 行)就比较简单了，许多字段，如中断请求等等，对于“PCI-PCI 桥”都是无意义的。所以对此只需要调用 pci_read_bases( )就可以了。同时，“PCI-PCI 桥”上可以有的存储区间数量也少，最多只能有两个。当然，实际上对“PCI-PCI 桥”的处理更为复杂，因为还得进一步扫描次层 PCI 总线，不过这是以后的事。

至此，对 pci_scan_device( )的调用已经完成，回到了 pci_scan_slot( )中，返回的是一个 pci_dev 结构指针，如果该指针为 0 则表示 PCI 总线上没有所指定的逻辑设备。接着的 pci_name_device( )将从设备中读入的厂商编号和设备编号转换成字符串。内核中为此设置了一个小小的数据库(转换表)，其内容见 drivers/pci/pci.ids，这里略举其中数行让读者有个印象：

```
1469        10b7  3Com Corporation
1470        0001  3c985 1000BaseSX
1471        3390  Token Link Velocity
1472        3590  3c359 TokenLink Velocity XL
```

意思是厂商编号 0x10b7 代表着 3Com，而若设备编号为 0x0001 则表示这是该公司生产的 3c985 网络卡。这个文件中包括了几乎所有知名的厂商和它们的产品。

接着，把 pci_scan_device( )返回的 pci_dev 数据结构挂入总的队列 pci_devices 以及该设备所在总线的队列。前面讲过，这个队列的头在代表着这个总线的 pci_bus 数据结构中。

本来，从一个逻辑设备读出头部信息并为之建立相应的数据结构以后，对这个设备的枚举就完成了。如前所述，这些信息是固化在芯片中的。可是，有些厂商在开始出售它们的某些产品以后却发现固化在里面的信息有错，因而需要在读出这些信息以后通过软件手段加以修正。有的修正需要在从设备读出头部信息后进行，有的则需要在设置了各个区间的总线地址以后进行。而厂商，则在出售产品时随同提供一段相应的程序。这样的情况不是一个两个，而是有许多。为此目的，Linux 内核中设计了一种统一的机制来进行这样的修正。首先是个数据结构(类型)pci_fixup，定义于 include/linux/pci.h：

```
660    /*
661     *  The world is not perfect and supplies us with broken PCI devices.
```

```
662      *  For at least a part of these bugs we need a work-around, so both
663      *  generic (drivers/pci/quirks.c) and per-architecture code can define
664      *  fixup hooks to be called for particular buggy devices.
665      */
666
667     struct pci_fixup {
668         int pass;
669         u16 vendor, device;     /* You can use PCI_ANY_ID here of course */
670         void (*hook)(struct pci_dev *dev);
671     };
```

这个数据结构表示：对于由厂商 vendor 提供的设备 device，需要在什么时候（pass）执行通过函数指针 hook 提供的函数。内核中有两个 pci_fixup 结构数组，提供了对已知厂商和产品的修正手段，分别定义于 arch/i386/kernel/pci-pc.c 和 drivers/pci/quirks.c 中。为篇幅的原因，我们从这两个结构数组中删去了许多元素，只留下几个，让读者有个印象。

```
927     struct pci_fixup pcibios_fixups[ ] = {
928         { PCI_FIXUP_HEADER, PCI_VENDOR_ID_INTEL,
                    PCI_DEVICE_ID_INTEL_82451NX,     pci_fixup_i450nx },
929         { PCI_FIXUP_HEADER, PCI_VENDOR_ID_INTEL,
                    PCI_DEVICE_ID_INTEL_82454GX,     pci_fixup_i450gx },
            . . . . . . .
938         { PCI_FIXUP_HEADER, PCI_VENDOR_ID_SI,
                    PCI_DEVICE_ID_SI_5598,       pci_fixup_latency },
939         { 0 }
940     };
```

```
250     /*
251      *  The main table of quirks.
252      */
253
254     static struct pci_fixup pci_fixups[ ]  __initdata = {
255         { PCI_FIXUP_FINAL,  PCI_VENDOR_ID_INTEL,
                    PCI_DEVICE_ID_INTEL_82441,  quirk_passive_release },
        . . . . . . .
283         { PCI_FIXUP_HEADER, PCI_VENDOR_ID_AL,
                    PCI_DEVICE_ID_AL_M7101,       quirk_ali7101_acpi },
284         { PCI_FIXUP_HEADER, PCI_VENDOR_ID_INTEL,
                    PCI_DEVICE_ID_INTEL_82371SB_2,  quirk_piix3_usb },
285         { PCI_FIXUP_HEADER, PCI_VENDOR_ID_INTEL,
                    PCI_DEVICE_ID_INTEL_82371AB_2,  quirk_piix3_usb },
286         { 0 }
287     };
288
```

例如，928 行说明：对于 Intel 的 82451NX 芯片(是“宿主—PCI 桥”)需要在读出了头部信息以后

调用 pci_fixup_i450nx( )进行修正 (PCI_FIXUP_HEADER 在 include/linux/pci.h 中定义为 1，PCI_FIXUP_FINAL 则为 2)。具体的修正是由 pci_fixup_device( )完成的，其代码在 drivers/pci/quirks.c 中，注意这里的调用参数 pass 为 PCI_FIXUP_HEADER，表示是读出头部信息以后的修正。

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( ) > pci_do_scan_bus( ) > pci_scan_slot( ) > pci_fixup_device( )]

```
305	void pci_fixup_device(int pass, struct pci_dev *dev)
306	{
307		pci_do_fixups(dev, pass, pcibios_fixups);
308		pci_do_fixups(dev, pass, pci_fixups);
309	}
```

这个函数对上述两个数组分别调用 pci_do_fixups( )，其代码在同一文件(quirks.c)中：

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( ) > pci_do_scan_bus( ) > pci_scan_slot( ) > pci_fixup_device( ) > pci_do_fixups( )]

```
290	static void pci_do_fixups(struct pci_dev *dev, int pass, struct pci_fixup *f)
291	{
292		while (f->pass) {
293			if (f->pass == pass &&
294			    (f->vendor == dev->vendor || f->vendor == (u16) PCI_ANY_ID) &&
295			    (f->device == dev->device || f->device == (u16) PCI_ANY_ID)){
296	#ifdef DEBUG
297				printk("PCI: Calling quirk %p for %s\n", f->hook, dev->slot_name);
298	#endif
299				f->hook(dev);
300			}
301			f++;
302		}
303	}
```

注意 293 行的条件，修正函数仅在指定的 pass 中才会执行。至于具体的修正函数，我们就不看了。

回到 pci_do_scan_bus( )的代码中，完成了对 pci_scan_slot( )的循环调用后，对一条 PCI 总线的扫描与枚举原则上已经完成了。但是，对得到的信息也可能需要作一些调整和修正。函数 pcibios_fixup_bus( )的代码在 arch/i386/kernel/pci-pc.c 中：

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( ) > pci_do_scan_bus( ) > pcibios_fixup_bus( )]

```
942	/*
943	 * Called after each bus is probed, but before its children
944	 * are examined.
945	 */
946
947	void __init pcibios_fixup_bus(struct pci_bus *b)
948	{
```

```
949         pcibios_fixup_ghosts(b);
950         pci_read_bridge_bases(b);
951     }
```

为什么要有这种调整和修正呢？原来，不光是 PCI 芯片中的信息会有错误，有些系统母板的设计也会发生错误。实践中发现，在有些母板上会在两个不同的综合地址(总是相距 16)中读出同一设备的头部信息，从而在枚举的过程中造成重复枚举，在所形成的 PCI 树中引入了“幻影”。所以，要通过 pcibios_fixup_ghosts( )进行一次扫描，把“幻影”去掉。这个函数的代码在 arch/i386/kernel/pci-pc.c 中，我们在这里就不看了。

函数 pci_read_bridge_bases( )的代码在 drivers/pci/pci.c 中：

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( ) > pci_do_scan_bus( ) > pcibios_fixup_bus( ) > pci_read_bridge_bases( )]

```
618     void __init pci_read_bridge_bases(struct pci_bus *child)
619     {
620         struct pci_dev *dev = child->self;
621         u8 io_base_lo, io_limit_lo;
622         u16 mem_base_lo, mem_limit_lo, io_base_hi, io_limit_hi;
623         u32 mem_base_hi, mem_limit_hi;
624         unsigned long base, limit;
625         struct resource *res;
626         int i;
627 
628         if (!dev)       /* It's a host bus, nothing to read */
629             return;
630 
631         for(i=0; i<3; i++)
632             child->resource[i] = &dev->resource[PCI_BRIDGE_RESOURCES+i];
633 
634         res = child->resource[0];
635         pci_read_config_byte(dev, PCI_IO_BASE, &io_base_lo);
636         pci_read_config_byte(dev, PCI_IO_LIMIT, &io_limit_lo);
637         pci_read_config_word(dev, PCI_IO_BASE_UPPER16, &io_base_hi);
638         pci_read_config_word(dev, PCI_IO_LIMIT_UPPER16, &io_limit_hi);
639         base = ((io_base_lo & PCI_IO_RANGE_MASK) << 8) | (io_base_hi << 16);
640         limit = ((io_limit_lo & PCI_IO_RANGE_MASK) << 8) | (io_limit_hi << 16);
641         if (base && base <= limit) {
642             res->flags = (io_base_lo & PCI_IO_RANGE_TYPE_MASK) | IORESOURCE_IO;
643             res->start = base;
644             res->end = limit + 0xfff;
645             res->name = child->name;
646         } else {
647             /*
648              * Ugh. We don't know enough about this bridge. Just assume
649              * that it's entirely transparent.
```

```
650             */
651           printk("Unknown bridge resource %d: assuming transparent\n", 0);
652           child->resource[0] = child->parent->resource[0];
653       }
654
655       res = child->resource[1];
656       pci_read_config_word(dev, PCI_MEMORY_BASE, &mem_base_lo);
657       pci_read_config_word(dev, PCI_MEMORY_LIMIT, &mem_limit_lo);
658       base = (mem_base_lo & PCI_MEMORY_RANGE_MASK) << 16;
659       limit = (mem_limit_lo & PCI_MEMORY_RANGE_MASK) << 16;
660       if (base && base <= limit) {
661           res->flags = (mem_base_lo & PCI_MEMORY_RANGE_TYPE_MASK) | IORESOURCE_MEM;
662           res->start = base;
663           res->end = limit + 0xfffff;
664           res->name = child->name;
665       } else {
666           /* See comment above. Same thing */
667           printk("Unknown bridge resource %d: assuming transparent\n", 1);
668           child->resource[1] = child->parent->resource[1];
669       }
670
671       res = child->resource[2];
672       pci_read_config_word(dev, PCI_PREF_MEMORY_BASE, &mem_base_lo);
673       pci_read_config_word(dev, PCI_PREF_MEMORY_LIMIT, &mem_limit_lo);
674       pci_read_config_dword(dev, PCI_PREF_BASE_UPPER32, &mem_base_hi);
675       pci_read_config_dword(dev, PCI_PREF_LIMIT_UPPER32, &mem_limit_hi);
676       base = (mem_base_lo & PCI_MEMORY_RANGE_MASK) << 16;
677       limit = (mem_limit_lo & PCI_MEMORY_RANGE_MASK) << 16;
678   #if BITS_PER_LONG == 64
679       base |= ((long) mem_base_hi) << 32;
680       limit |= ((long) mem_limit_hi) << 32;
681   #else
682       if (mem_base_hi || mem_limit_hi) {
683           printk(KERN_ERR
        "PCI: Unable to handle 64-bit address space for %s\n", child->name);
684           return;
685       }
686   #endif
687       if (base && base <= limit) {
688           res->flags = (mem_base_lo & PCI_MEMORY_RANGE_TYPE_MASK) |
                      IORESOURCE_MEM | IORESOURCE_PREFETCH;
689           res->start = base;
690           res->end = limit + 0xfffff;
691           res->name = child->name;
692       } else {
693           /* See comments above */
694           printk("Unknown bridge resource %d: assuming transparent\n", 2);
695           child->resource[2] = child->parent->resource[2];
```

```
696             }
697     }
```

PCI 桥的配置寄存器组与一般 PCI 设备的不同。如前所述，一般 PCI 设备可以有 6 个地址区间，外加一个 ROM 区间，代表着设备上实际存在的存储器或寄存器区间；而 PCI 桥，则本身并不一定有存储器或寄存器区间，但是却有三个用于地址过滤的区间。每个地址过滤区间决定了一个地址窗口，从靠近 CPU 一侧发出的地址，如果落在 PCI 桥的某个窗口之内，就可以穿过 PCI 桥而到达其所连接的总线上。反过来，从总线一侧由 PCI 设备发出的地址(如果有的话)，则要在相应 PCI 桥的所有窗口之外才能穿过 PCI 桥到达靠近 CPU 一侧。此外，PCI 桥的命令寄存器中还有“memory access enable”和“I/O access enable”两个控制位，当这两个控制位为 0 时，这些窗口就全都关上了(两个方向都关上)。在未完成对 PCI 总线的初始化之前，还没有为 PCI 设备上的各个区间分配合适的总线地址时，正是因为这两个控制位为 0，才不会对 CPU 一侧造成干扰。

显然，每个窗口的大小和位置应该是总线上各个相应区间的总和，而总线上的各个区间则应该基本上互相连续并且互不重叠。因此，一个窗口实际上也是一个区间，所以也用一个 resource 数据结构表示。不过，说是各个相应区间的总和，实际上窗口一定是按某种边界对齐的，其大小可能会大于实际的需要(从而造成少量浪费)。

第一个区间是 I/O 地址的窗口。PCI 桥上有两个 8 位寄存器，即 PCI_IO_BASE 和 PCI_IO_LIMIT，用来确定具体窗口的起点和终点。其中高 4 位就是 16 位 I/O 地址中的最高 4 位，低 4 位则固定为 0(对 16 位 I/O 地址而言，见下)。窗口的大小为 4KB 的倍数，并与 4KB 边界对齐。其起点为 PCI_IO_BASE 的高 4 位后面添上 12 位 0，终点则为 PCI_IO_LIMIT 的高 4 位后面添上 12 位 1。例如，要是 PCI_IO_BASE 的高 4 位为 6 而 PCI_IO_LIMIT 的高 4 位为 7，就表示窗口的范围为 0x6000 至 0x7fff。PCI 总线并不是专为 i386 处理器设计的，有些处理器可能使用 32 位 I/O 地址空间。所以，如果具体的 PCI 设备支持 32 位 I/O 地址，则进一步通过 PCI_IO_BASE_UPPER16 和 PCI_IO_LIMIT_UPPER16 两个 16 位寄存器提供窗口起点和终点的高 16 位，并且使 PCI_IO_BASE 和 PCI_IO_LIMIT 中的低 4 位固定为 1。

第二个区间是存储器地址的窗口。寄存器 PCI_MEMORY_BASE 和 PCI_MEMORY_LIMIT 的构造与作用跟上述 PCI_IO_BASE 和 PCI_IO_LIMIT 相似，只不过是 16 位寄存器，除最低 4 位为 0 以外，其高 12 位为 32 位存储器地址中的最高 12 位。窗口的大小则为 1MB 的倍数，并与 1MB 边界对齐。例如，要是 PCI_MEMORY_BASE 的高 12 位为 0xa81 而 PCI_MEMORY_LIMIT 的高 12 位也是 0xa81，就表示窗口的范围为 0xa8100000～0xa81fffff，共 1MB。这个区间主要用于映射在存储器地址空间的 I/O 寄存器。

第三个区间是“可预取”存储器地址的窗口。寄存器 PCI_PREF_MEMORY_BASE 和 PCI_PREF_MEMORY_LIMIT 的构造与作用跟 PCI_MEMORY_BASE 和 PCI_MEMORY_LIMIT 相似，但是其最低 4 位并不总是 0，而是以 0 表示 32 位地址，以 1 表示 64 位地址。对于 64 位地址，PCI 桥上还有 PCI_PREF_BASE_UPPER32 和 PCI_PREF_LIMIT_UPPER32 两个 32 位寄存器用于地址的高 32 位。

回到 pci_do_scan_bus( )的代码中，现在一条 PCI 总线上的所有设备都已经枚举完毕了。但是，其中有些设备可能是“PCI-PCI 桥”，对这些设备还得“顺藤摸瓜”进一步扫描相应的次层 PCI 总线。对此，代码中分两趟(见 996 行)扫描当前 PCI 总线的队列 devices，如果发现一个设备的类型为“PCI-PCI 桥”或“PCI-CardBus 桥”(999 行)，就对其调用 pci_scan_bridge( )，这个函数的代码在 drivers/pci/pci.c

中。那么，为什么要分两趟扫描呢？这是因为在两趟扫描中针对的情况是不同的，第一趟扫描是针对已经由 BIOS 进行过处理的 PCI 桥，第一趟扫描则是针对未经 BIOS 处理的 PCI 桥。

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( ) > pci_do_scan_bus( ) > pci_scan_bridge( )]

```
746    /*
747     * If it's a bridge, configure it and scan the bus behind it.
748     * For CardBus bridges, we don't scan behind as the devices will
749     * be handled by the bridge driver itself.
750     *
751     * We need to process bridges in two passes -- first we scan those
752     * already configured by the BIOS and after we are done with all of
753     * them, we proceed to assigning numbers to the remaining buses in
754     * order to avoid overlaps between old and new bus numbers.
755     */
756    static int __init pci_scan_bridge(struct pci_bus *bus,
                                        struct pci_dev * dev, int max, int pass)
757    {
758        unsigned int buses;
759        unsigned short cr;
760        struct pci_bus *child;
761        int is_cardbus = (dev->hdr_type == PCI_HEADER_TYPE_CARDBUS);
762
763        pci_read_config_dword(dev, PCI_PRIMARY_BUS, &buses);
764        DBG("Scanning behind PCI bridge %s, config %06x, pass %d\n",
                                        dev->slot_name, buses & 0xffffff, pass);
765        if ((buses & 0xffff00) && !pcibios_assign_all_busses( )) {
766            /*
767             * Bus already configured by firmware, process it in the first
768             * pass and just note the configuration.
769             */
770            if (pass)
771                return max;
772            child = pci_add_new_bus(bus, dev, 0);
773            child->primary = buses & 0xFF;
774            child->secondary = (buses >> 8) & 0xFF;
775            child->subordinate = (buses >> 16) & 0xFF;
776            child->number = child->secondary;
777            if (!is_cardbus) {
778                unsigned int cmax = pci_do_scan_bus(child);
779                if (cmax > max) max = cmax;
780            } else {
781                unsigned int cmax = child->subordinate;
782                if (cmax > max) max = cmax;
783            }
784        } else {
785            /*
```

```
786             * We need to assign a number to this bus which we always
787             * do in the second pass. We also keep all address decoders
788             * on the bridge disabled during scanning.  FIXME: Why?
789             */
790            if (!pass)
791                return max;
792            pci_read_config_word(dev, PCI_COMMAND, &cr);
793            pci_write_config_word(dev, PCI_COMMAND, 0x0000);
794            pci_write_config_word(dev, PCI_STATUS, 0xffff);
795
796            child = pci_add_new_bus(bus, dev, ++max);
797            buses = (buses & 0xff000000)
798                  | ((unsigned int)(child->primary)     <<  0)
799                  | ((unsigned int)(child->secondary)   <<  8)
800                  | ((unsigned int)(child->subordinate) << 16);
801            /*
802             * We need to blast all three values with a single write.
803             */
804            pci_write_config_dword(dev, PCI_PRIMARY_BUS, buses);
805            if (!is_cardbus) {
806                /* Now we can scan all subordinate buses... */
807                max = pci_do_scan_bus(child);
808            } else {
809                /*
810                 * For CardBus bridges, we leave 4 bus numbers
811                 * as cards with a PCI-to-PCI bridge can be
812                 * inserted later.
813                 */
814                max += 3;
815            }
816            /*
817             * Set the subordinate bus number to its real value.
818             */
819            child->subordinate = max;
820            pci_write_config_byte(dev, PCI_SUBORDINATE_BUS, max);
821            pci_write_config_word(dev, PCI_COMMAND, cr);
822        }
823        sprintf(child->name,
             (is_cardbus ? "PCI CardBus #%02x" : "PCI Bus #%02x"), child->number);
824        return max;
825    }
```

首先从 PCI 桥的配置寄存器组中读出含有“主总线号”、“次总线号”等字段的长字，其中最低的字节为主总线号，在此之上的两个字节则依次为次总线号和子树中的最大总线号。目前，pcibios_assign_all_busses( )是个空操作，固定地定义为 0，所以只要两个较高字节不为全 0 就能满足 765 行的条件。这两个字节不为全 0 说明在此之前这条总线已经枚举过一次，所以已经分配了次总线号或最大“子总线号”，一般而言这发生于系统加电时由 PCI BIOS 进行的扫描枚举。对于这样的情况，下

面 772～783 行的代码只在第一趟扫描时执行。

既然是发现了一条新的 PCI 总线，当然就要为之建立一个 pci_bus 数据结构。当然，BIOS 在扫描中必定也建立了某种形式的数据结构，记录了相应的信息，但是在这里内核要撇开 BIOS，独立地建立其自己的数据结构。函数 pci_add_new_bus( )的代码在 drivers/pci/pci.c 中：

[pci_init( ) > pcibios_init( ) > pci_scan_bus ( ) > pci_do_scan_bus( ) > pci_scan_bridge( ) > pci_add_new_bus( )]

```
712     static struct pci_bus * __init pci_add_new_bus(struct pci_bus *parent,
                                                       struct pci_dev *dev, int busnr)
713     {
714         struct pci_bus *child;
715         int i;
716
717         /*
718          * Allocate a new bus, and inherit stuff from the parent..
719          */
720         child = pci_alloc_bus( );
721
722         list_add_tail(&child->node, &parent->children);
723         child->self = dev;
724         dev->subordinate = child;
725         child->parent = parent;
726         child->ops = parent->ops;
727         child->sysdata = parent->sysdata;
728
729         /*
730          * Set up the primary, secondary and subordinate
731          * bus numbers.
732          */
733         child->number = child->secondary = busnr;
734         child->primary = parent->secondary;
735         child->subordinate = 0xff;
736
737         /* Set up default resource pointers.. */
738         for (i = 0; i < 4; i++)
739             child->resource[i] = &dev->resource[PCI_BRIDGE_RESOURCES+i];
740
741         return child;
742     }
```

我们把这段代码留给读者。

回到上面 pci_scan_bridge( )的代码中，在进一步设置了新的 pci_bus 数据结构以后，就对这条次层 PCI 总线调用 pci_do_scan_bus( )进行扫描枚举。显然，这是对 pci_do_scan_bus( )的递归调用，因为 CPU 此时正是在这个函数的下面执行。可见，这是在对系统的 PCI 总线结构作深度优先的搜索。

再看当 765 行的条件不能满足，即首次扫描枚举一条次层 PCI 总线时的操作，那就是代码中的 785～

821 行。这一段代码仅在第二趟扫描时执行。对于未经 BIOS 处理的 PCI 桥，第一趟扫描的目的只在于知道总线号已经用到了多大，这样才可以在第二趟扫描中在此基础上继续分配总线号，并将分配的编号设置进该“PCI-PCI 桥”的配置寄存器组。这里先读入命令寄存器的内容并保存起来，然后往命令寄存器写入全 0，往状态寄存器写入全 1(二者均为 16 位寄存器)。接着通过 pci_add_new_bus( )为其建立一个 pci_bus 数据结构，注意这里调用 pci_add_new_bus( )时的第三个参数 max 为次层总线的编号，这就是当前的最大总线号，并且先已加 1。然后，797 行拼组起包含着此总线号的长字，并在 804 行将其写入配置寄存器组。同样地，对于“PCI-PCI 桥”要递归调用 pci_do_scan_bus( )，这个函数返回已经在该子树中使用的最大总线号，820 行将其写入配置寄存器组中的 PCI_SUBORDINATE_BUS 字节。最后，821 行恢复命令寄存器原来的内容。

回到 pcibios_init( )的代码中(987 行)。对 0 号 PCI 总线的(递归的)扫描枚举本身已经完成了，系统已经部分地(在大多数情况下实际上是全部)知道了自己的“家底”，下面就可以统筹地进行对设备的设置了。这种设置包括两方面的内容，其一是设备的中断请求线与系统中的中断控制器之间的连接，其二是设备上各个区间在总线上的地址映射。如前所述，这就是从前要通过跳线或小开关设置的两大内容。不过，实际上很可能 BIOS 已经在系统加电自检的阶段已经进行了设置，所以往往只是加以确认而已。另一方面，系统中除 0 号总线之外还可能有其他主总线存在，也需要加以扫描，但是下面读者就会看到：出于效率的考虑，这种扫描要推迟到处理中断请求线时才进行。

前面讲过，在 PCI 总线上有 INTA-INTD 共 4 条中断请求线。凡是能产生中断请求的 PCI 设备，其中断请求必定连接在其中的一条上，此时其寄存器 PCI_INTERRUPT_PIN 的内容(1～4)表示连接在哪一条线上。按 PCI 总线规格书的规定，凡是单功能 PCI 模块(接口卡)的中断请求都应该连接在 INTA 上，多功能模块(多个逻辑设备共存于同一物理模块上)才可以在使用 INTA 之余再使用其他的中断请求线。那么，这 INTA～INTD 4 条中断请求线又怎样连接到中断控制器的中断请求输入线呢？PCI 总线规格书对此并没有作硬性的规定，给具体系统的设计和实现留下了变通的余地。通常，系统的中断控制器一共有 16 条中断请求输入线，每个 PCI 设备的中断请求线(如果有的话)就是要选择连接到其中的一条上去。理想的情况当然是每条中断请求输入线最多只连接一个中断源，但是一般而言这是难以办到的，实际上只能对各项设备加以适当的组合，让若干设备共用同一条中断请求输入线。配备有 PCI 总线的 PC 机母板一般都采用一种可编程中断请求路径互连器(router，通常与 PCI-ISA 桥集成在同一芯片中)，由软件设置 4 条 PCI 中断请求线与中断控制器的中断请求输入线之间的互连，图 8.4 是一种典型设计的示意图。

从图中可以看出，在 PCI 接口卡上将中断请求都连接在 INTA 上其实只有局限于具体 PCI 插槽的意义，只是使接口卡的结构比较划一而已。实际上，(这种)系统母板的设计自动将各种设备的中断请求分布到了路径互连器的各条输入线上。从系统软件的角度，我们关切的有两个方面。一是选择、确定，并通过路径互连器实施互连；二是要搞清楚具体插槽中具体设备的中断请求到底连接到了中断控制器的哪一条输入线上，这样才能确定应该把设备的中断服务程序“登记”到哪一个中断服务程序队列中(见第 3 章)。为达到这个目的，BIOS 通常提供一个“中断路径表”，为各条 PCI 总线的各个插槽提供其 4 条中断请求线的去向，就是在母板上连接到了路径互连器的哪条输入线上。至于路径互连器的输出，则总是一对一地连接到中断控制器上。

可见，对于 PCI 总线中断机制的初始化，中断路径表和路径互连器是两个关键。事实上 pcibios_irq_init( )正是从中断路径表开始的，其代码在 arch/i386/kernel/pci-irq.c 中。

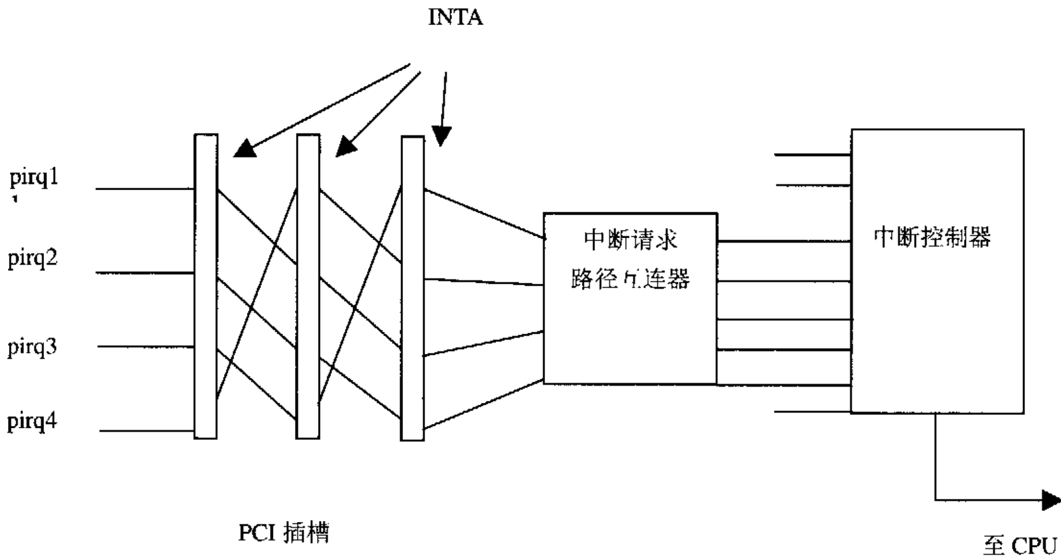


图 8.4　PCI 总线上中断请求线的互连

[pci_init( ) > pcibios_init( ) > pcibios_irq_init( )]

```
516    void __init pcibios_irq_init(void)
517    {
518        DBG("PCI: IRQ init\n");
519        pirq_table = pirq_find_routing_table( );
520    #ifdef CONFIG_PCI_BIOS
521        if (!pirq_table && (pci_probe & PCI_BIOS_IRQ_SCAN))
522            pirq_table = pcibios_get_irq_routing_table( );
523    #endif
524        if (pirq_table) {
525            pirq_peer_trick( );
526            pirq_find_router( );
527            if (pirq_table->exclusive_irqs) {
528                int i;
529                for (i=0; i<16; i++)
530                    if (!(pirq_table->exclusive_irqs & (1 << i)))
531                        pirq_penalty[i] += 100;
532            }
533      /* If we're using the I/O APIC, avoid using the PCI IRQ routing table */
534            if (io_apic_assign_pci_irqs)
535                pirq_table = NULL;
536        }
537    }
```

如上所述，PCI 总线的 4 根中断请求线与路径互连器的连接是因母板的设计而异的，所以由具体母板上的 BIOS 提供一个 PCI “中断请求路径表”，其数据结构定义于 arch/i386/kernel/pci-i386.h:

```
54      struct irq_routing_table {
55          u32 signature;          /* PIRQ_SIGNATURE should be here */
56          u16 version;            /* PIRQ_VERSION */
57          u16 size;           /* Table size in bytes */
58          u8 rtr_bus, rtr_devfn;      /* Where the interrupt router lies */
59          u16 exclusive_irqs;     /* IRQs devoted exclusively to PCI usage */
60          u16 rtr_vendor, rtr_device; /* Vendor and device ID of interrupt router */
61          u32 miniport_data;      /* Crap */
62          u8 rfu[11];
63          u8 checksum;            /* Modulo 256 checksum must give zero */
64          struct irq_info slots[0];
65      } __attribute__((packed));
```

路径表的第一个长字一定是一个特殊的数值 PIRQ_SIGNATURE，实际上是 4 个特殊的字符，版本号则必须是 PIRQ_VERSION，均定义于 arch/i386/kernel/pci-i386.h 中。路径表的起点一定是与 16 字节边界对齐的，但并不固定在某个特定的位置上，所以需要扫描寻找。

```
22      #define PIRQ_SIGNATURE  (('$'<< 0) + ('P'<< 8) + ('I'<< 16) + ('R'<< 24))
23      #define PIRQ_VERSION 0x0100
```

这个 irq_routing_table 数据结构其实只是中断请求路径表的头部，表的主体 slots 是一个 irq_info 结构数组，这种数据结构的定义在 arch/i386/kernel/pci-i386.h 中：

```
44      struct irq_info {
45          u8 bus, devfn;          /* Bus, device and function */
46          struct {
47              u8 link;        /* IRQ line ID, chipset dependent, 0=not routed */
48              u16 bitmap;     /* Available IRQs */
49          } __attribute__((packed)) irq[4];
50          u8 slot;            /* Slot number, 0=onboard */
51          u8 rfu;
52      } __attribute__((packed));
```

对于系统中每条 PCI 总线上的每个模块，路径表中都有个 irq_info 结构。结构中给出了其 4 条中断请求线与路径互连器输入线的连接，同时还有个位图，说明了可供选择的连接对象，即中断控制器的各条输入线。

首先通过 pirq_find_routing_table( )在 BIOS 所在的区间扫描，以找到中断请求路径表，其代码在 arch/i386/kernel/pci-irq.c 中：

[pci_init( ) > pcibios_init( ) > pcibios_irq_init ( ) > pirq_find_routing_table( )]

```
46      /*
47       *  Search 0xf0000 -- 0xfffff for the PCI IRQ Routing Table.
48       */
49
```

```
50    static struct irq_routing_table * __init pirq_find_routing_table(void)
51    {
52        u8 *addr;
53        struct irq_routing_table *rt;
54        int i;
55        u8 sum;
56
57        for(addr = (u8 *) __va(0xf0000); addr < (u8 *) __va(0x100000);
                                                        addr += 16) {
58            rt = (struct irq_routing_table *) addr;
59            if (rt->signature != PIRQ_SIGNATURE ||
60                rt->version != PIRQ_VERSION ||
61                rt->size % 16 ||
62                rt->size < sizeof(struct irq_routing_table))
63                continue;
64            sum = 0;
65            for(i=0; i<rt->size; i++)
66                sum += addr[i];
67            if (!sum) {
68                DBG("PCI: Interrupt Routing Table found at 0x%p\n", rt);
69                return rt;
70            }
71        }
72        return NULL;
73    }
```

内核中有个全局的指针 pirq_table，指向从 BIOS 找到的中断请求路径表。如果找到了这个路径表(524 行)，就要进一步加以处理了。函数 pirq_peer_trick( )的代码在 arch/i386/kernel/pci-irq.c 中：

[pci_init( ) > pcibios_init( ) > pcibios_irq_init( ) > pirq_peer_trick( )]

```
75    /*
76     * If we have a IRQ routing table, use it to search for peer host
77     * bridges.  It's a gross hack, but since there are no other known
78     * ways how to get a list of buses, we have to go this way.
79     */
80
81    static void __init pirq_peer_trick(void)
82    {
83        struct irq_routing_table *rt = pirq_table;
84        u8 busmap[256];
85        int i;
86        struct irq_info *e;
87
88        memset(busmap, 0, sizeof(busmap));
89        for(i=0;
      i < (rt->size - sizeof(struct irq_routing_table)) / sizeof(struct irq_info); i++)
```

```
        {
90              e = &rt->slots[i];
91      #ifdef DEBUG
92              {
93                  int j;
94                  DBG("%02x:%02x slot=%02x", e->bus, e->devfn/8, e->slot);
95                  for(j=0; j<4; j++)
96                      DBG(" %d:%02x/%04x",j,e->irq[j].link, e->irq[j].bitmap);
97                  DBG("\n");
98              }
99      #endif
100             busmap[e->bus] = 1;
101         }
102         for(i=1; i<256; i++)
103        /*
104         *  It might be a secondary bus, but in this case its parent is already
105         *  known (ascending bus order) and therefore pci_scan bus returns
                 immediately.
106         */
107             if (busmap[i] && pci_scan_bus(i, pci_root bus->ops, NULL))
108                 printk("PCI: Discovered primary peer bus %02x [IRQ]\n", i);
109         pcibios_last_bus = -1;
110     }
```

代码中先通过 89 行的 for 循环对中断请求路径表中各个表项所涉及的总线作一番统计，循环结束以后，凡是与数组 busmap[ ](以总线号为下标)中为 1 的表项相对应的总线都是在路径表中有表项的。一般而言，这些总线应该都已经在前一阶段中完成了扫描枚举，但是从 pcibios_init( )中通过 pci_scan_bus( )扫描的是从 0 号总线开始的一棵树，如果系统中除此之外还有其他的 PCI 主总线，则尚未受到扫描。现在，既然有了一份所有 PCI 总线的清单，就可以在其指引下有目标地通过 pci_scan_bus( )扫描。这个函数对于已经扫描的 PCI 总线实际上不起作用，并返回 0(读者最好到有关的代码中验证一下)；而若返回非 0，则说明扫描到了一条未经扫描的 PCI 总线(更确切地说是一棵树)，而这必定是一条主总线。最后，代码中把一个全局量 pcibios_last_bus 设成－1，表示系统中所有的主总线都已扫描，以后就不用再为此操心了。

回到 pcibios_irq_init( )的代码中，补上了可能的遗漏以后，接着就要来处理路径互连器了。中断路径互连器通常与 PCI-ISA 桥集成在同一芯片中，并且也作为 PCI 设备连接在 PCI 总线上。路径表头部的 rtr_bus 和 rtr_devfn 两个字段指明了该设备所在的位置，rtr_vendor,和 rtr_device 两个字段则为芯片的提供者及其产品编号。函数 pirq_find_router( )的作用就是找到这个设备的 pci_dev 数据结构，并根据其提供者及产品编号找到相应的驱动函数，其代码也在 arch/i386/kernel/pci-irq.c 中：

[pci_init( ) > pcibios_init( ) > pcibios_irq_init( ) > pirq_find_router( )]

```
347     static struct irq_router *pirq_router;
348     static struct pci_dev *pirq_router dev;
349
```

```
350     static void __init pirq_find_router(void)
351     {
352         struct irq_routing_table *rt = pirq_table;
353         struct irq_router *r;
354
355     #ifdef CONFIG_PCI_BIOS
        ......
361     #endif
362         /* fall back to default router if nothing else found */
363         pirq_router = pirq_routers +
                    sizeof(pirq_routers) / sizeof(pirq_routers[0]) - 1;
364
365         pirq_router_dev = pci_find_slot(rt->rtr_bus, rt->rtr_devfn);
366         if (!pirq_router_dev) {
367             DBG("PCI: Interrupt router not found at %02x:%02x\n",
                                            rt->rtr_bus, rt->rtr_devfn);
368             return;
369         }
370
371         for(r=pirq_routers; r->vendor; r++) {
372             /* Exact match against router table entry? Use it! */
373             if (r->vendor == rt->rtr_vendor && r->device == rt->rtr_device) {
374                 pirq_router = r;
375                 break;
376             }
377             /* Match against router device entry? Use it as a fallback */
378             if (r->vendor == pirq_router_dev->vendor &&
                    r->device == pirq_router_dev->device) {
379                 pirq_router = r;
380             }
381         }
382         printk("PCI: Using IRQ router %s [%04x/%04x] at %s\n",
383             pirq_router->name,
384             pirq_router_dev->vendor,
385             pirq_router_dev->device,
386             pirq_router_dev->slot_name);
387     }
```

中断路径互连器由一个 irq_router 数据结构代表，定义于 arch/i386/kernel/pci-irq.c：

```
39      struct irq_router {
40          char *name;
41          u16 vendor, device;
42          int (*get)(struct pci_dev *router, struct pci_dev *dev, int pirq);
43          int (*set)(struct pci_dev *router, struct pci_dev *dev, int pirq, int new);
44      };
```

结构中的函数指针 get 和 set 用于读出和改变芯片中的互连，不同的芯片可能有不同的 get 和 set 函数。PC 机中可能采用的此类芯片有好几种，arch/i386/kernel/pci-irq.c 中定义了一个 irq_router 结构数组 pirq_routers[ ]，里面包括了各种常用的芯片。为减小篇幅，我们从中删去了一部分。

```
323	static struct irq_router pirq_routers[ ] = {
324		{ "PIIX", PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82371FB_0,
						pirq_piix_get, pirq_piix_set },
325		{ "PIIX", PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82371SB_0,
						pirq_piix_get, pirq_piix_set },
		......
330		{ "ALI", PCI_VENDOR_ID_AL, PCI_DEVICE_ID_AL_M1533,
						pirq_ali_get, pirq_ali_set },
331		{ "VIA", PCI_VENDOR_ID_VIA, PCI_DEVICE_ID_VIA_82C586_0,
						pirq_via_get, pirq_via_set },
		......
336		{ "OPTI", PCI_VENDOR_ID_OPTI, PCI_DEVICE_ID_OPTI_82C700,
						pirq_opti_get, pirq_opti_set },
337		{ "NatSemi", PCI_VENDOR_ID_CYRIX, PCI_DEVICE_ID_CYRIX_5520,
						pirq_cyrix_get, pirq_cyrix_set },
		......
344		{ "default", 0, 0, NULL, NULL }
345	};
```

内核中有个全局的指针 pirq_router，用来指向中断路径互连器的 irq_router 数据结构。先通过 pci_find_slot( )找到其 pci_dev 数据结构。如果找不到就让指针 pirq_router 指向 pirq_routers[ ]中的最后一项“default”，即不存在 get 和 set 两个函数。否则就在 pirq_routers[ ]中找到所用的中断路径互连器芯片，并使 pirq_router 指向相应的 irq_router 数据结构。

此外，路径表中还有个位图 exclusive_irqs，位图中为 1 的位表示中断控制器的相应输入应该应该专用，而避免为多个中断源所共用。所以，对于这些中断请求输入线，代码中在一个数组 pirq_penalty[ ] 的相应元素上增加了一个“惩罚量”100，使得在选择中断路径互连时不太会选择这些作为对象。

还要注意，中断路径互连器仅在采用常规的 8259A 中断控制器时才使用，所以若系统采用“高级可编程中断控制器”APIC 就在前面的 535 行把全局指针 pirq_table 设成 0，因为 APIC 本来就是“可编程”的。

那么，要是在 BIOS 中找不到中断请求路径表或者找不到路径互连器怎么办呢？那就只要采用默认的连接，不需要设置。

前面，在 pirq_peer_trick( )中，根据中断路径表的指引对系统中除 0 号总线以外的主总线进行了有目标的补充扫描。但是，如果根本就没有路径表呢？那就只好进行穷举式的扫描了。当然，穷举扫描的效率比有目标的扫描要低，这就是为什么把对其他主总线的扫描推迟到找到了(或找不到) 中断路径表以后的原因。当然，其代价是使代码更难懂了。

函数 pcibios_fixup_peer_bridges( )对除 0 号总线以外的主总线进行穷举扫描。但是如果已经在路径表的指引下进行了扫描，则全局量 pcibios_last_bus 为负(见前面 pirq_peer_trick( )的代码)，因而立即就返回。这个函数的代码在 arch/i386/kernel/pci-pc.c 中：

[pci_init( ) > pcibios_init( ) > pcibios_fixup_peer_bridges( )]

```
776    /*
777     * Discover remaining PCI buses in case there are peer host bridges.
778     * We use the number of last PCI bus provided by the PCI BIOS.
779     */
780    static void __init pcibios_fixup_peer_bridges(void)
781    {
782        int n;
783        struct pci_bus bus;
784        struct pci_dev dev;
785        u16 l;
786
787        if (pcibios_last_bus <= 0 || pcibios_last_bus >= 0xff)
788            return;
789        DBG("PCI: Peer bridge fixup\n");
790        for (n=0; n <= pcibios_last_bus; n++) {
791            if (pci_bus_exists(&pci_root_buses, n))
792                continue;
793            bus.number = n;
794            bus.ops = pci_root_ops;
795            dev.bus = &bus;
796            for(dev.devfn=0; dev.devfn<256; dev.devfn += 8)
797                if (!pci_read_config_word(&dev, PCI_VENDOR_ID, &l) &&
798                    l != 0x0000 && l != 0xffff) {
799                DBG("Found device at %02x:%02x [%04x]\n", n, dev.devfn, l);
800                    printk("PCI: Discovered peer bus %02x\n", n);
801                    pci_scan_bus(n, pci_root_ops, NULL);
802                    break;
803                }
804        }
805    }
```

我们把这个函数留给读者。

回到 pcibios_init( )的代码中，我们继续阅读有关中断请求路径互连的代码，函数 pcibios_fixup_irqs( )的代码在 arch/i386/kernel/pci-irq.c 中，我们抽去了采用 APIC 时的条件编译部分。

[pci_init( ) > pcibios_init( ) > pcibios_fixup_irqs( )]

```
539    void __init pcibios_fixup_irqs(void)
540    {
541        struct pci_dev *dev;
542        u8 pin;
543
544        DBG("PCI: IRQ fixup\n");
545        pci_for_each_dev(dev) {
546            /*
```

```
547              * If the BIOS has set an out of range IRQ number, just ignore it.
548              * Also keep track of which IRQ's are already in use.
549              */
550             if (dev->irq >= 16) {
551                 DBG("%s: ignoring bogus IRQ %d\n", dev->slot_name, dev->irq);
552                 dev->irq = 0;
553             }
554             /* If the IRQ is already assigned to a PCI device,
                                      ignore its ISA use penalty */
555             if (pirq_penalty[dev->irq] >= 100 && pirq_penalty[dev->irq] < 100000)
556                 pirq_penalty[dev->irq] = 0;
557             pirq_penalty[dev->irq]++;
558         }
559
560         pci_for_each_dev(dev) {
561             pci_read_config_byte(dev, PCI_INTERRUPT_PIN, &pin);
562     #ifdef CONFIG_X86_IO_APIC
        . . . . . .
595     #endif
596             /*
597              * Still no IRQ? Try to lookup one...
598              */
599             if (pin && !dev->irq)
600                 pcibios_lookup_irq(dev, 0);
601         }
602     }
```

这里的 pci_for_each_dev( )是个宏定义，是个对全局 pci_dev 数据结构队列 pci_devices 的循环，定义于 include/linux/pci.h：

```
305     #define pci_for_each_dev(dev) \
306         for(dev = pci_dev_g(pci_devices.next); \
                dev != pci_dev_g(&pci_devices); \
                dev = pci_dev_g(dev->global_list.next))
```

如前所述，每个中断请求最终都要连接到中断控制器的某条中断请求输入线上。不过，中断控制器的 16 条输入线并不是可以随意选用的，例如 0 号中断请求线就应该由时钟中断专用。另一方面，还要尽可能把各项 PCI 设备均匀地分布到不同的中断请求线上。为此，内核中设立了一个(整数)数组 pirq_penalty[ ]，与一个简单的算法结合在一起，就可以确定中断请求输入线的选择，这个数组定义于 arch/i386/kernel/pci-irq.c 中：

```
27      /*
28       * Never use: 0, 1, 2 (timer, keyboard, and cascade)
29       * Avoid using: 13, 14 and 15 (FP error and IDE).
30       * Penalize: 3, 4, 6, 7, 12 (known ISA uses: serial, floppy, parallel and mouse)
31       */
```

```
32    unsigned int pcibios_irq_mask = 0xfff8;
33
34    static int pirq_penalty[16] = {
35        1000000, 1000000, 1000000, 1000, 1000, 0, 1000, 1000,
36        0, 0, 0, 0, 1000, 100000, 100000, 100000
37    };
```

数组的大小为 16，对应着 16 条中断请求输入线。每个元素代表着因选用相应中断请求输入线而受到的“惩罚”(penalty)，所以选择时总是要选用受惩罚最小的。中断请求输入线 0、1、2、13、14、和 15 的惩罚一开始就是 1000000，所以实际上永远不会选用这几条输入线。一开始时，中断请求输入线 5、8、9、10、和 11 的惩罚为 0，所以开始时总是在这几条线中选择。对于同一条中断请求输入线，选用它的设备愈多，则再次选用它时受到的“惩罚”就愈大。所以，只要设备的数量不是太大，则中断请求输入线 3、4、6、7 和 12 也实际上不会被选用。而负荷的均匀分布，则在选择的过程中自然得到了保证。

代码中通过一个 for 循环(545 行)扫描系统中所有的 PCI 设备，对使用中断控制器各条输入线的设备计数(557 行)，从而在 pirq_penalty[ ]中累积起选用各条中断请求输入线的惩罚量。中断控制器的有些输入线最好能保留给 ISA 总线上的设备专用，路径表中通过一个位图指明了这些输入线。为此，前面 pcibios_irq_init( )中在 pirq_penalty[ ]的相应元素上增加了一个惩罚量 100。可是，那只是为了让 PCI 设备避免使用这些输入线，如果发现某个 PCI 设备已经在使用这样的输入线，那就又另当别论了。既然已经有 PCI 设备在用，戒规已经打破，那就不妨让别的 PCI 设备也来使用，所以这里(556 行)干脆把惩罚量设成 0，重新计数。

然后，又通过 for 循环再来一次对所有 PCI 设备的扫描(560 行)。如果从寄存器 PCI_INTERRUPT_PIN 读入的内容非 0，就表示该设备有中断功能。此时如果 dev->irq 为 0，即尚不知道与中断控制器的哪一条中断请求输入线相连，就要通过 pcibios_lookup_irq( )寻找。寻找什么呢？如前所述，如果知道一个设备的中断请求连到了路径互连器的哪一条输入线，并且发现这条线已经在路径互连器中连接到了中断控制器，也就知道了或者说发现了该设备的中断请求的最终去向。但是，如果在路径互连器中尚未连接呢？此时是否要在中断控制器的输入线中作出选择并完成连接呢？这是由第二个调用参数决定的。这里(600 行)，把参数设成 0，表示如果尚未连接就留着再说。这个函数的代码在 arch/i386/kernel/pci-irq.c 中，这个函数比较长，我们分段阅读。

[pci_init( ) > pcibios_init( ) > pcibios_fixup_irqs( ) > pcibios_lookup_irq( )]

```
405    static int pcibios_lookup_irq(struct pci_dev *dev, int assign)
406    {
407        u8 pin;
408        struct irq_info *info;
409        int i, pirq, newirq;
410        int irq = 0;
411        u32 mask;
412        struct irq_router *r = pirq_router;
413        struct pci_dev *dev2;
414        char *msg = NULL;
```

```
415
416         if (!pirq_table)
417             return 0;
418
419         /* Find IRQ routing entry */
420         pci_read_config_byte(dev, PCI_INTERRUPT_PIN, &pin);
421         if (!pin) {
422             DBG(" -> no interrupt pin\n");
423             return 0;
424         }
425         pin = pin - 1;
426
427         DBG("IRQ for %s:%d", dev->slot_name, pin);
428         info = pirq_get_info(dev);
429         if (!info) {
430             DBG(" -> not found in routing table\n");
431             return 0;
432         }
433         pirq = info->irq[pin].link;
434         mask = info->irq[pin].bitmap;
435         if (!pirq) {
436             DBG(" -> not routed\n");
437             return 0;
438         }
439         DBG(" -> PIRQ %02x, mask %04x, excl %04x",
                                pirq, mask, pirq_table->exclusive_irqs);
440         mask &= pcibios_irq_mask;
441
442         /*
443          * Find the best IRQ to assign: use the one
444          * reported by the device if possible.
445          */
446         newirq = dev->irq;
447         if (!newirq && assign) {
448             for (i = 0; i < 16; i++) {
449                 if (!(mask & (1 << i)))
450                     continue;
451                 if (pirq_penalty[i] < pirq_penalty[newirq] &&
452        !request_irq(i, pcibios_test_irq_handler, SA_SHIRQ, "pci-test", dev)){
453                     free_irq(i, dev);
454                     newirq = i;
455                 }
456             }
457         }
458         DBG(" -> newirq=%d", newirq);
459
```

先从设备的配置寄存器组读入寄存器 PCI_INTERRUPT_PIN，以得知其中断请求连在 PCI 总线的

哪一条中断请求线上。这个数值是从 1 开始的，所以要把它调整成从 0 开始(425 行)。然后，通过 pirq_get_info( )从 BIOS 提供的中断路径表中找到所在总线和插槽的路径信息，这个函数的代码在 arch/i386/kernel/pci-irq.c 中：

[pci_init( ) > pcibios_init( ) > pcibios_fixup_irqs( ) > pcibios_lookup_irq( ) > pirq_get_info( )]

```
389    static struct irq_info *pirq_get_info(struct pci_dev *dev)
390    {
391        struct irq_routing_table *rt = pirq_table;
392        int entries = (rt->size -
                   sizeof(struct irq_routing_table)) / sizeof(struct irq_info);
393        struct irq_info *info;
394
395        for (info = rt->slots; entries--; info++)
396            if (info->bus == dev->bus->number &&
                       PCI_SLOT(info->devfn) == PCI_SLOT(dev->devfn))
397                return info;
398        return NULL;
399    }
```

如果根据总线号和插槽号找到了相应的 irq_info 数据结构，那么这个结构中的数组 irq[ ]记录着这个特定插槽的 4 条中断请求线跟中断路径互连器的连接。对于每条中断请求线，数据结构中提供了两个字段。其中 link 的主体是路径互连器的输入线号；bitmap 则是一个位图，表示哪一些中断控制器的输入是可以选择作为目标的。在 bitmap 的基础上，内核中还有个全局的屏蔽位图 pcibios_irq_mask，定义为 0xfff8，表示中断请求号 0、1、2 是不能选用的(440 行)。

我们继续往下看 pcibios_lookup_irq( ) 的代码(arch/i386/kernel/pci-irq.c)。

[pci_init( ) > pcibios_init( ) > pcibios_fixup_irqs( ) > pcibios_lookup_irq( )]

```
460        /* Check if it is hardcoded */
461        if ((pirq & 0xf0) == 0xf0) {
462            irq = pirq & 0xf;
463            DBG(" -> hardcoded IRQ %d\n", irq);
464            msg = "Hardcoded";
465            if (dev->irq && dev->irq != irq) {
466          printk("IRQ routing conflict in pirq table! Try 'pci=autoirq'\n");
467                return 0;
468            }
469        } else if (r->get && (irq = r->get(pirq_router_dev, dev, pirq))) {
470            DBG(" -> got IRQ %d\n", irq);
471            msg = "Found";
472         /* We refuse to override the dev->irq information. Give a warning! */
473                if (dev->irq && dev->irq != irq) {
474            printk("IRQ routing conflict in pirq table! Try 'pci=autoirq'\n");
475                    return 0;
476                }
```

```
477         } else if (newirq && r->set &&
                            (dev->class >> 8) != PCI_CLASS_DISPLAY_VGA) {
478             DBG(" -> assigning IRQ %d", newirq);
479             if (r->set(pirq_router_dev, dev, pirq, newirq)) {
480                 eisa_set_level_irq(newirq);
481                 DBG(" ... OK\n");
482                 msg = "Assigned";
483                 irq = newirq;
484             }
485         }
486 
487         if (!irq) {
488             DBG(" ... failed\n");
489             if (newirq && mask == (1 << newirq)) {
490                 msg = "Guessed";
491                 irq = newirq;
492             } else
493                 return 0;
494         }
495         printk("PCI: %s IRQ %d for device %s\n", msg, irq, dev->slot_name);
496 
```

在由路径表提供的信息中，字段 link 的高 4 位为 1 时表示路径互连器内部的连接是“硬连接”，此时其低 4 位就是中断控制器的输入线号；否则便表示路径互连器内部的连接可以通过 get 和 set 两个函数指针读出或设置。所以，对于一般的路径互连器，只要其函数指针 get 非 0，就可以通过它读出连接的目标。如果这个函数的返回值(irq)非 0，那就是所连接的目标，否则说明尚未连接。尚未连接又怎么办呢？前面讲过了，留着再说，后面读者会看到究竟留到什么时候再说。

如果已经连接，则我们已经搞清楚了目标设备中断请求的最终去向，把这个信息记录在 dev->irq 中就可以了。但是，系统中可能有很多 PCI 设备都连在同一条中断请求线上，因而有着相同的去向，应该与这些设备分享这个信息。我们继续往下看 pcibios_lookup_irq( )的代码(arch/i386/kernel/pci-irq.c)。

[pci_init( ) > pcibios_init( ) > pcibios_fixup_irqs( ) > pcibios_lookup_irq( )]

```
497         /* Update IRQ for all devices with the same pirq value */
498         pci_for_each_dev(dev2) {
499             pci_read_config_byte(dev2, PCI_INTERRUPT_PIN, &pin);
500             if (!pin)
501                 continue;
502             pin--;
503             info = pirq_get_info(dev2);
504             if (!info)
505                 continue;
506             if (info->irq[pin].link == pirq) {
507                 dev2->irq = irq;
508                 pirq_penalty[irq]++;
509                 if (dev != dev2)
```

```
510                     printk("PCI: The same IRQ used for device %s\n",
                                                        dev2->slot_name);
511             }
512         }
513         return 1;
514     }
```

这段代码就很简单了。

搞清了中断请求的去向，下一步就是为各个 PCI 设备中的各个地址区间分配总线地址，并设置好各个 PCI 设备对这些区间(到总线地址)的映射了。我们在前面已经看到，每个 PCI 设备都通过配置寄存器组提供其各个区间的起始地址和区间大小，但那可能只是在设备内部的地址，或者是由 BIOS 分配的总线地址。所以，对于前者需要在一个统一的总线地址空间为这些区间分配地址并建立映射，对于后者则要加以验证和确认，并为之建立起相应的数据结构。对于 CPU 来说，总线地址相当于物理地址，以后在此基础上还要再加上一层映射，将虚拟地址映射到总线地址。在分配的过程中，只要原来已经分配的地址可用，就尽量维持原状，对这些地址只是验证一下，并为之建立起相应的数据结构。那么，什么样的地址才是不可用的从而需要另行分配的呢？后面读者将会看到详情，这里先简单说一下。首先，在系统初始化以后，已经把物理地址空间低端的一大块分配给了内核本身，这些地址当然不能再用于 PCI 总线，而设备内部的地址都在低端，因此需要为之另行分配地址。其次，PCI 设备的各个区间不允许互相冲突，如果发生冲突也要作出调整。对于 I/O 地址也是一样。

对于存储器和 I/O 两种地址资源的管理，内核中有一套地址资源管理机制。每一个逻辑意义上独立的连续地址区间都以一个 resource 数据结构代表，定义于 include/linux/ioport.h：

```
11    /*
12     * Resources are tree-like, allowing
13     * nesting etc..
14     */
15    struct resource {
16        const char *name;
17        unsigned long start, end;
18        unsigned long flags;
19        struct resource *parent, *sibling, *child;
20    };
```

结构中的 start 和 end 表示该区间的地址范围，flags 是表示区间性质的一些标志位。指针 child、sibling 和 parent 则用来维系可以上下两个方向攀援的树形结构。每个区间(的 resource 结构)都通过指针 child 指向其第一个子区间，而同一区间的所有子区间则通过指针 sibling 形成一个单链，并都通过指针 parent 指向其父区间。例如，假定有一块 2KB 的地址空间 0x1000～0x17ff，这区间已经分配给某项特定的用途，假定是在一块 4 口串行接口卡上用作接收缓冲区，所以已经有了一个 resource 数据结构。然后，对于此项特定的用途，又把其中 0x1000～0x11ff、0x1200～0x13ff 和 0x1400～0x15ff 三个 512 字节的子区间进一步分配给了这块卡上的三个串行口，但是还剩下最后 512 字节尚未分配出去，那么在 2KB 的 resource 数据结构下面就有一个 3 个 resource 数据结构的队列，如图 8.5 所示。以后，当要为第 4 个

串行口分配缓冲区时，先在 2KB 的 resource 数据结构中找到该区间的顶端为 0x17ff，而在其子区间队列中则找到已经分配的最高地址为 0x15ff，并且在此之前并无合适的空洞可供分配，所 c 正好可以把最后 512 字节分配出去，并且在其子区间队列中再挂上一个 0x1600～0x17ff 的 resource 数据结构。

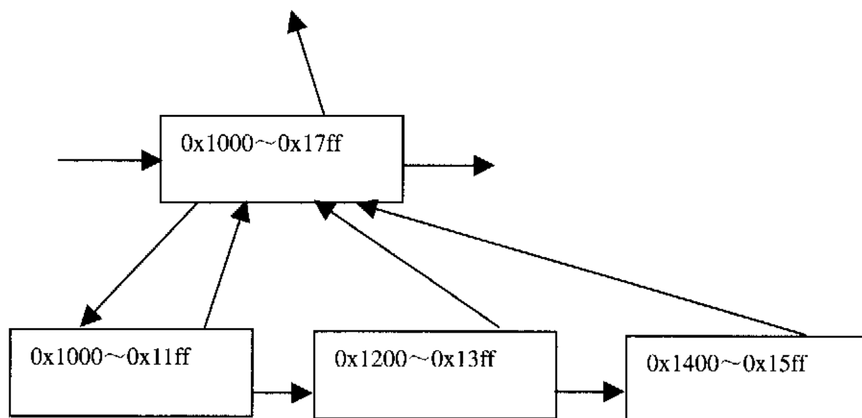


图 8.5　地址空间分配示意图

最初时，系统中只有两个区间，一个代表着 I/O 地址空间，另一个代表着内存地址。前面讲过，对于 I/O 地址空间只能通过 I/O 指令访问，而对内存地址空间则只能通过访内指令访问。这两个区间的 resource 结构定义于 kernel/resource.c：

```
18      struct resource ioport_resource = { "PCI IO", 0x0000,
                                           IO_SPACE_LIMIT, IORESOURCE_IO };
19      struct resource iomem_resource = { "PCI mem", 0x00000000,
                                           0xffffffff, IORESOURCE_MEM };
```

我们在前面也已看到，在为主 PCI 总线建立 pci_bus 结构时，它的两个区间指针一个指向 ioport_resource，表示如果需要 I/O 地址区间就从 ioport_resource 中分配；另一个指向 iomem_resource，表示如果需要内存地址区间就从 iomem_resource 中分配。不过，系统在初始化阶段已经从这两个空间中分配了许多资源，所以已经不再是像它们的初值所表示的整个 I/O 地址空间或整个内存地址空间了。

对总线地址的确认与分配是由 pcibios_resource_survey( ) 完成的，其代码在 arch/i386/kernel/pci-i386.c 中：

[pci_init( ) > pcibios_init( ) > pcibios_resource_survey ( )]

```
297     void __init pcibios_resource_survey(void)
298     {
299         DBG("PCI: Allocating resources\n");
300         pcibios_allocate_bus_resources(&pci_root_buses);
301         pcibios_allocate_resources(0);
302         pcibios_allocate_resources(1);
303         pcibios_assign_resources( );
304     }
```

首先通过 pcibios_allocate_bus_resources( )为每条 PCI 总线分配地址资源，其代码在同一文件(pci-i386.c)中：

[pci_init( ) > pcibios_init( ) > pcibios_resource_survey ( ) > pcibios_allocate_bus_resources( )]

```
152    /*
153     * Handle resources of PCI devices.  If the world were perfect, we could
154     * just allocate all the resource regions and do nothing more.  It isn't.
155     * On the other hand, we cannot just re-allocate all devices, as it would
156     * require us to know lots of host bridge internals.  So we attempt to
157     * keep as much of the original configuration as possible, but tweak it
158     * when it's found to be wrong.
159     *
160     * Known BIOS problems we have to work around:
161     * - I/O or memory regions not configured
162     * - regions configured, but not enabled in the command register
163     * - bogus I/O addresses above 64K used
164     * - expansion ROMs left enabled (this may sound harmless, but given
165     *   the fact the PCI specs explicitly allow address decoders to be
166     *   shared between expansion ROMs and other resource regions, it's
167     *   at least dangerous)
168     *
169     * Our solution:
170     * (1) Allocate resources for all buses behind PCI-to-PCI bridges.
171     *     This gives us fixed barriers on where we can allocate.
172     * (2) Allocate resources for all enabled devices.  If there is
173     *     a collision, just mark the resource as unallocated. Also
174     *     disable expansion ROMs during this step.
175     * (3) Try to allocate resources for disabled devices.  If the
176     *     resources were assigned correctly, everything goes well,
177     *     if they weren't, they won't disturb allocation of other
178     *     resources.
179     * (4) Assign new addresses to resources which were either
180     *     not configured at all or misconfigured.  If explicitly
181     *     requested by the user, configure expansion ROM address
182     *     as well.
183     */
184
185    static void __init pcibios_allocate_bus_resources(struct list_head *bus_list)
186    {
187        struct list_head *ln;
188        struct pci_bus *bus;
189        struct pci_dev *dev;
190        int idx;
191        struct resource *r, *pr;
192
193        /* Depth-First Search on bus tree */
```

```
194         for (ln=bus_list->next; ln != bus_list; ln=ln->next) {
195             bus = pci_bus_b(ln);
196             if ((dev = bus->self)) {
197                 for (idx = PCI_BRIDGE_RESOURCES; idx < PCI_NUM_RESOURCES; idx++) {
198                     r = &dev->resource[idx];
199                     if (!r->start)
200                         continue;
201                     pr = pci_find_parent_resource(dev, r);
202                     if (!pr || request_resource(pr, r) < 0)
203                         printk(KERN_ERR "PCI: Cannot allocate resource region %d of
                            bridge %s\n", idx, dev->slot_name);
204                 }
205             }
206             pcibios_allocate_bus_resources(&bus->children);
207         }
208     }
```

参数 bus_list 指向一个队列头，实际上是一个 pci_bus 结构队列，第一次调用这个函数时指向队列 pci_root_buses，即所有通过“宿主—PCI 桥”相连的 PCI 总线(通常只有一条)。如前所述，每条 PCI 总线都有可能通过“PCI-PCI 桥”连接到更次层的其他 PCI 总线，所以每个 pci_bus 数据结构中都有个队列头 children，用来维持次层 PCI 总线的 pci_bus 结构队列。完成了对 PCI 总线和设备的枚举以后，这些数据结构都已经各就各位了。既然 PCI 总线的系统结构是递归的，对整个 PCI 结构的资源分配就自然也是递归的，所以 206 行对次层 pci_bus 结构队列递归调用 pcibios_allocate_bus_resources( )，向下作深度优先的遍历。代码中外层的 for 循环(194 行)是对同一队列中所有 pci_bus 结构的循环。对于总线本身，即 PCI 桥，由内层的 for 循环(197 行)对其 4 个(从 7 至 10)地址区间加以检验。

这里的常数 PCI_BRIDGE_RESOURCES 和 PCI_NUM_RESOURCES 均定义于 include/linux/pci.h:

```
367     /*
368      *  For PCI devices, the region numbers are assigned this way:
369      *
370      *  0-5 standard PCI regions
371      *  6   expansion ROM
372      *  7-10    bridges: address space assigned to buses behind the bridge
373      */
374
375     #define PCI_ROM_RESOURCE 6
376     #define PCI_BRIDGE_RESOURCES 7
377     #define PCI_NUM_RESOURCES 11
```

对于普通的 PCI 设备，其 pci_dev 结构中的开头 6 个(0 至 5)地址区间是设备上可能有的区间，第 7 个区间(6)是可能有的扩充 ROM 区间。如果这设备是个 PCI 桥，则后面还有 4 个区间，pci_bus 结构中的 4 个 resource 指针就分别指向这 4 个区间。前面，我们在阅读 pci_read_bridge_bases( )时说过，PCI 桥本身并不“使用”这些区间中的地址，而是用这些区间作为地址过滤的窗口。其中第一个窗口用于 I/O 地址，第二个用于存储器地址，第三个则为“可预取” 存储器地址区间。此外，还有一个用于扩

充 ROM 区间的窗口。次层总线上所有设备(包括 PCI 桥)所使用的地址都必须在这些窗口中.换言之，这些设备所需要的地址资源都要从这些区间中分配。所以，每个 PCI 桥或者说每条 PCI 总线，都需要从其上层“批发”下一些地址资源，然后“零售”分配给连接在这条总线上的所有设备，包括把其中的一部分批发给更次层总线。就这样，每一条 PCI 总线上的设备都向其所在的总线批发地址资源，而总线则向其上层总线批发。那么，顶层的 PCI 总线又向谁批发呢？那就是 ioport_resource 和 iomem_resource，这是两种地址资源终极的来源。

如果 PCI 桥的某个区间已经有了对资源的需求，就要先通过 pci_find_parent_resource( )看看其“父节点”是否拥有其所需的地址资源，其代码在 include/linux/pci.h 中：

[pci_init( ) > pcibios_init( ) > pcibios_resource_survey ( ) > pcibios_allocate_bus_resources( ) > pci_find_parent_resource( )]

```
164    /**
165     * pci_find_parent_resource - return resource region of parent bus of given region
166     * @dev: PCI device structure contains resources to be searched
167     * @res: child resource record for which parent is sought
168     *
169     *  For given resource region of given device, return the resource
170     *  region of parent bus the given region is contained in or where
171     *  it should be allocated from.
172     */
173    struct resource *
174    pci_find_parent_resource(const struct pci_dev *dev, struct resource *res)
175    {
176        const struct pci_bus *bus = dev->bus;
177        int i;
178        struct resource *best = NULL;
179
180        for(i=0; i<4; i++) {
181            struct resource *r = bus->resource[i];
182            if (!r)
183                continue;
184            if (res->start && !(res->start >= r->start && res->end <= r->end))
185                continue;   /* Not contained */
186            if ((res->flags ^ r->flags) & (IORESOURCE_IO | IORESOURCE_MEM))
187                continue;   /* Wrong type */
188            if (!((res->flags ^ r->flags) & IORESOURCE_PREFETCH))
189                return r;   /* Exact match */
190            if ((res->flags & IORESOURCE_PREFETCH) && !(r->flags & IORESOURCE_PREFETCH))
191                best = r; /* Approximating prefetchable by non-prefetchable */
192        }
193        return best;
194    }
```

参数 dev 指向代表着次层总线的 PCI 桥的 pci_dev 数据结构，res 则指向代表着所需地址区间的 resource 结构。分配时依次扫描 PCI 桥所在总线的 4 个地址区间。

分配的大原则是地址的范围必须相符(185 行)，并且类型(存储器地址或 I/O 地址)必须相符(187 行)。其次，是否“可预取”也最好能一致。要不然，如果所要求的区间用于“可预取”的存储器，而总线上的区间本来是供不可预取的寄存器使用的，那么虽然有些勉强，也还不失为一个选择(191 和 193 行，反过来就不行了)。

如果在父节点中找到了能够满足要求的区间，则函数返回指向该区间的指针(否则为 0)，说明所需的地址资源是有保障的，所以就通过 request_resource( )加以分配，这个函数的代码在 kernel/resource.c 中：

[pci_init( ) > pcibios_init( ) > pcibios_resource_survey ( ) > pcibios_allocate_bus_resources( ) > request_resource ( )]

```
114    int request_resource(struct resource *root, struct resource *new)
115    {
116        struct resource *conflict;
117
118        write_lock(&resource_lock);
119        conflict = __request_resource(root, new);
120        write_unlock(&resource_lock);
121        return conflict ? -EBUSY : 0;
122    }
```

落实资源分配的过程涉及队列操作，不容许受到打扰，所以必须加锁。具体的操作则由 __request_resource( )完成，其代码也在同一文件中：

[pci_init( ) > pcibios_init( ) > pcibios_resource_survey ( ) > pcibios_allocate_bus_resources( ) > request_resource ( ) > __request_resource( )]

```
66    /* Return the conflict entry if you can't request it */
67    static struct resource * __request_resource(struct resource *root,
                                                   struct resource *new)
68    {
69        unsigned long start = new->start;
70        unsigned long end = new->end;
71        struct resource *tmp, **p;
72
73        if (end < start)
74            return root;
75        if (start < root->start)
76            return root;
77        if (end > root->end)
78            return root;
79        p = &root->child;
80        for (;;) {
81            tmp = *p;
82            if (!tmp || tmp->start > end) {
83                new->sibling = tmp;
```

```
84                  *p = new;
85                  new->parent = root;
86                  return NULL;
87              }
88              p = &tmp->sibling;
89              if (tmp->end < start)
90                  continue;
91              return tmp;
92          }
93      }
```

这里的参数 root 指向总线的某个 resource 数据结构，而 new 则指向 PCI 桥，即次层总线的 resource 数据结构。代码中通过一个 for 循环在 root 的子区间队列中为 new 找到适当的位置，然后将 new 插入该队列中。如果发现 new 与已经存在的子区间冲突，则操作失败而返回与其冲突的区间指针。

当完成了对 pcibios_allocate_bus_resources( )的递归调用时，所有 PCI 总线所需的地址资源都已经分配好了。但是，读者大概也看出来了，除对于冲突的检验以外，这里所谓"分配"其实只是一种"事后追认"而已。这些区间的范围本来就是从 PCI 桥中读出来的，现在也并未加以改变。所做的只是为这些区间建立起了 resource 结构，并插入到整个地址资源树中的某个位置上，以供检验冲突之用。那么，这些 PCI 桥是怎么知道应该有什么样的窗口的呢？这是由 BIOS 设置的。既然已经设置，也就没有必要推倒重来。如果 BIOS 并未设置，那也不难通过扫描已经建立起的 pci_bus 和 pci_dev 数据结构统计出来，或者在扫描、枚举的过程中计算出来。

回到 pcibios_resource_survey( )的代码中，接着就可以为所有的 PCI 设备分配地址资源了，这里分两趟调用 pcibios_allocate_resources( )，这个函数的代码在 arch/i386/kernel/pci-i386.c 中。同样，所谓"分配"实际上往往只是追认。

[pci_init( ) > pcibios_init( ) > pcibios_resource_survey ( ) > pcibios_allocate_resources( )]

```
210     static void __init pcibios_allocate_resources(int pass)
211     {
212         struct pci_dev *dev;
213         int idx, disabled;
214         u16 command;
215         struct resource *r, *pr;
216
217         pci_for_each_dev(dev) {
218             pci_read_config_word(dev, PCI_COMMAND, &command);
219             for(idx = 0; idx < 6; idx++) {
220                 r = &dev->resource[idx];
221                 if (r->parent)          /* Already allocated */
222                     continue;
223                 if (!r->start)          /* Address not assigned at all */
224                     continue;
225                 if (r->flags & IORESOURCE_IO)
226                     disabled = !(command & PCI_COMMAND_IO);
227                 else
```

```
228                         disabled = !(command & PCI_COMMAND_MEMORY);
229                     if (pass == disabled) {
230                         DBG("PCI: Resource %08lx-%08lx (f=%lx, d=%d, p=%d)\n",
231                             r->start, r->end, r->flags, disabled, pass);
232                         pr = pci_find_parent_resource(dev, r);
233                         if (!pr || request_resource(pr, r) < 0) {
234     printk(KERN_ERR "PCI: Cannot allocate resource region %d of device %s\n",
                                                            idx, dev->slot_name);
235                             /* We'll assign a new address later */
236                             r->end -= r->start;
237                             r->start = 0;
238                         }
239                     }
240                 }
241                 if (!pass) {
242                     r = &dev->resource[PCI_ROM_RESOURCE];
243                     if (r->flags & PCI_ROM_ADDRESS_ENABLE) {
244                         /* Turn the ROM off, leave the resource region,
                                                but keep it unregistered. */
245                         u32 reg;
246                         DBG("PCI: Switching off ROM of %s\n", dev->slot_name);
247                         r->flags &= ~PCI_ROM_ADDRESS_ENABLE;
248                         pci_read_config_dword(dev, dev->rom_base_reg, &reg);
249                         pci_write_config_dword(dev, dev->rom_base_reg,
                                                    reg & ~PCI_ROM_ADDRESS_ENABLE);
250                     }
251                 }
252             }
253     }
```

这是一个对所有pci_dev数据结构的循环。对于每一项PCI设备的 6 个常规区间，如果这些区间已经生效(可以接受访问)就在第一趟扫描(pass为 0)时分配地址资源，否则就在第二趟扫描(pass为 1)时分配。对扩充ROM区间则只在第一趟，即参数pass为 0 时加以处理。

PCI 设备的这些区间有几种可能。第一种是已经分配了地址资源，因而其指针 parent 已指向其父节点，对这种节点无需作任何处理，所以把它跳过就行了(222 行)。第二种是区间的起始地址为 0，这种区间或者是本来就不需要为之分配空间，或者是已知其父节点暂时不能满足其需要，这里也把它跳过(224 行)。第三种就是需要分配地址资源的了，如果通过 pci_find_parent_resource( )发现这部分地址资源已经包含在父节点的资源中，那就可以通过 request_resource( )分配了，这两个函数的代码我们都已在前面读过。如果发现不能在当前的起始地址上从父节点分配到所需的地址资源，即地址范围不符或者发生了冲突，就先将该区间平移到起始地址为 0 的地方(236～237 行)，这些区间的起始地址需要加以变更才行。这里要说明，对这些区间的地址资源分配之所以失败并不是由于父节点中的资源短缺，而是因为对起始地址的要求不能满足。BIOS 在确定父节点的窗口大小时是经过计算的，加上对窗口位置的对齐，父节点的窗口一般都要比实际的需要大。所以，只要允许将区间适当平移，就一定能分配到所需的地址资源。

对于 ROM 区间，则在第一趟扫描时予以关闭。ROM 区间一般只是在初始化时由 BIOS 或具体的设备驱动程序使用，所以现在可以关闭了。但是其地址资源暂时还保存着，如果需要还可以在设备驱动程序中再打开。

对于不能在原有起始地址上分配到所需地址资源的区间，即起始地址已改成 0 的区间，要通过 pcibios_assign_resources( )加以分配，实际上这才是真正意义上的“分配”而不是追认。这个函数的代码在 arch/i386/kernel/pci-i386.c 中：

[pci_init( ) > pcibios_init( ) > pcibios_resource_survey ( ) > pcibios_assign_resources( )]

```
255    static void __init pcibios_assign_resources(void)
256    {
257        struct pci_dev *dev;
258        int idx;
259        struct resource *r;
260
261        pci_for_each_dev(dev) {
262            int class = dev->class >> 8;
263
264            /* Don't touch classless devices and host bridges */
265            if (!class || class == PCI_CLASS_BRIDGE_HOST)
266                continue;
267
268            for(idx=0; idx<6; idx++) {
269                r = &dev->resource[idx];
270
271                /*
272                 *  Don't touch IDE controllers and I/O ports of video cards!
273                 */
274                if ((class == PCI_CLASS_STORAGE_IDE && idx < 4) ||
275                    (class == PCI_CLASS_DISPLAY_VGA &&
                                (r->flags & IORESOURCE_IO)))
276                    continue;
277
278                /*
279                 *  We shall assign a new address to this resource, either because
280                 *  the BIOS forgot to do so or because we have decided the old
281                 *  address was unusable for some reason.
282                 */
283                if (!r->start && r->end)
284                    pci_assign_resource(dev, idx);
285            }
286
287            if (pci_probe & PCI_ASSIGN_ROMS) {
288                r = &dev->resource[PCI_ROM_RESOURCE];
289                r->end -= r->start;
290                r->start = 0;
```

```
291                 if (r->end)
292                     pci_assign_resource(dev, PCI_ROM_RESOURCE);
293             }
294         }
295     }
```

这个函数就是个嵌套的 for 循环，对于系统中的所有 PCI 设备，只要这个设备有效(类型字段非 0)，并且不是 PCI 桥，就在内层循环中检查其 6 个可能的地址区间。只要是需要分配地址资源的地址区间(起始地址为 0 而终点地址非 0)，便通过 pci_assign_resource ( )为其分配总线地址，并将其设置入具体设备的配置寄存器组。回顾一下前面 pcibios_allocate_resources( )的代码，就可以看出这些区间正是在那里没有能解决的区间。当时之所以没有能解决是因为指定了起始地址，现在则放宽了条件。

不过，IDE 存储设备(硬盘)的前 4 个区间和 VGA 显示设备的 I/O 地址区间是特例，这些区间的地址不需要分配、也不能改变(因为已经在用了)。

函数 pci_assign_resource ( )的代码在 drivers/pci/setup-res.c 中：

[pci_init( ) > pcibios_init( ) > pcibios_resource_survey ( ) > pcibios_assign_resources( )
 > pci_assign_resource( )]

```
99     int
100    pci_assign_resource(struct pci_dev *dev, int i)
101    {
102        const struct pci_bus *bus = dev->bus;
103        struct resource *res = dev->resource + i;
104        unsigned long size, min;
105
106        size = res->end - res->start + 1;
107        min = (res->flags & IORESOURCE_IO) ? PCIBIOS_MIN_IO : PCIBIOS_MIN_MEM;
108
109        /* First, try exact prefetching match.. */
110        if (pci_assign_bus_resource(bus, dev, res, size,
                                       min, IORESOURCE_PREFETCH, i) < 0) {
111            /*
112             * That failed.
113             *
114             * But a prefetching area can handle a non-prefetching
115             * window (it will just not perform as well).
116             */
117            if (!(res->flags & IORESOURCE_PREFETCH) ||
                   pci_assign_bus_resource(bus, dev, res, size, min, 0, i) < 0)
               {
118            printk(KERN_ERR
                   "PCI: Failed to allocate resource %d for %s\n", i, dev->name);
119                return -EBUSY;
120            }
121        }
122
```

```
123         DBGC(("  got res[%lx:%lx] for resource %d of %s\n", res->start,
124                             res->end, i, dev->name));
125
126         return 0;
127     }
```

这个函数通过 pci_assign_bus_resource( )为给定设备从其所在的总线分配地址区间。为设备上的每个区间分配地址时，一方面要考虑区间的实际大小，另一方面对其位置也有所限制，对于 I/O 地址区间其起点不得低于 PCIBIOS_MIN_IO，对于内存地址区间则不得低于 PCIBIOS_MIN_MEM，这两个常数均定义于 include/asm-i386/pci.h：

```
12     #define PCIBIOS_MIN_IO         0x1000
13     #define PCIBIOS_MIN_MEM        0x10000000
```

就是说，I/O 地址区间的位置不得低于 4KB，而内存地址区间的位置不得低于 256MB。另一方面，对于地址区间还有个是否必须满足“可预取”要求的问题，所以这里(110 行)先以 IORESOURCE_PREFETCH 为参数调用一次 pci_assign_bus_resource( )，表示要求在“可预取”方面相符。若不能满足要求，则根据情况再以 0 为参数调用一次(117 行)，表示在这方面不相符也可以，但这只适用于设备上的区间为“可预取”，即用于存储器的情况下，反过来就不可以了。函数 pci_assign_bus_resource( )的代码也在 drivers/pci/setup-res.c 中：

[pci_init( ) > pcibios_init( ) > pcibios_resource_survey ( ) > pcibios_assign_resources( )
 > pci_assign_resource( ) > pci_assign_bus_resource( )]

```
59     /*
60      * Given the PCI bus a device resides on, try to
61      * find an acceptable resource allocation for a
62      * specific device resource..
63      */
64     static int pci_assign_bus_resource(const struct pci_bus *bus,
65         struct pci_dev *dev,
66         struct resource *res,
67         unsigned long size,
68         unsigned long min,
69         unsigned int type_mask,
70         int resno)
71     {
72         int i;
73
74         type_mask |= IORESOURCE_IO | IORESOURCE_MEM;
75         for (i = 0 ; i < 4; i++) {
76             struct resource *r = bus->resource[i];
77             if (!r)
78                 continue;
79
```

```
80          /* type mask must match */
81          if ((res->flags ^ r->flags) & type_mask)
82              continue;
83
84          /* We cannot allocate a non-prefetching resource
                                        from a pre-fetching area */
85          if ((r->flags & IORESOURCE_PREFETCH) && !(res->flags & IORESOURCE_PREFETCH))
86              continue;
87
88          /* Ok, try it out.. */
89          if (allocate_resource(r, res, size, min, -1, size,
                                    pcibios_align_resource, dev) < 0)
90              continue;
91
92          /* Update PCI config space. */
93          pcibios_update_resource(dev, r, res, resno);
94          return 0;
95      }
96      return -EBUSY;
97  }
```

从哪里分配地址区间呢？设备在哪一条 PCI 总线上，就从已经成块“批发”给这条总线的地址区间中分配。这里的参数 bus 指向总线的 pci_bus 结构，而 res 则指向一个 resource 数据结构，结构中记载着对目标区间的要求。代码中通过一个 for 循环依次尝试该总线的 4 个可能的区间。如果二者类型相符，即同为 I/O 地址或同为内存地址，并且是否“可预取”也相符(如果要求的话)，就通过 allocate_resource( )为其分配总线地址 (kernel/resource.c)，如果分配成功就通过 pcibios_update_resource( ) 将其设置进目标设备。

[pci_init( ) > pcibios_init( ) > pcibios_resource_survey ( ) > pcibios_assign_resources( ) > pci_assign_resource( ) > pci_assign_bus_resource( ) > allocate_resource( )]

```
185  /*
186   * Allocate empty slot in the resource tree given range and alignment.
187   */
188  int allocate_resource(struct resource *root, struct resource *new,
189                    unsigned long size,
190                    unsigned long min, unsigned long max,
191                    unsigned long align,
192                    void (*alignf)(void *, struct resource *, unsigned long),
193                    void *alignf_data)
194  {
195      int err;
196
197      write_lock(&resource_lock);
198      err = find_resource(root, new, size, min, max, align, alignf, alignf_data);
199      if (err >= 0 &&  request_resource(root, new))
```

```
200                 err = -EBUSY;
201             write_unlock(&resource_lock);
202             return err;
203     }
```

参数 root 指向一个 resource 数据结构，代表着设备所在总线上的一个地址区间；而 new 则指向另一个 resource 数据结构，代表着待分配的地址区间。这块地址的大小由参数 size 确定，实际上一定是 2 的某次幂，位置须在 min 与 max 之间，align 则与边界对齐有关。对照前面对这个函数的调用，就可以看出这里 size 是实际的区间大小，min 为 PCIBIOS_MIN_IO 或 PCIBIOS_MIN_MEM，max 为 0xffffffff，而 align 与 size 相同。还有，参数 alignf 是个函数指针，实际上指向 pcibios_align_resource( )，alignf_data 则实际上指向目标设备的 pci_dev 数据结构。注意这里的 max 是个无符号整数，所以－1 实际上是全 1。这些参数原封不动地传给 find_resource( )，先通过这个函数找到符合要求的区间，其代码在 kernel/resource.c 中：

[pci_init( ) > pcibios_init( ) > pcibios_resource_survey ( ) > pcibios_assign_resources( )
 > pci_assign_resource( ) > pci_assign_bus_resource( ) > allocate_resource( )
 > find_resource( )]

```
148     /*
149      * Find empty slot in the resource tree given range and alignment.
150      */
151     static int find_resource(struct resource *root, struct resource *new,
152                     unsigned long size,
153                     unsigned long min, unsigned long max,
154                     unsigned long align,
155                     void (*alignf)(void *, struct resource *, unsigned long),
156                     void *alignf_data)
157     {
158         struct resource *this = root->child;
159
160         new->start = root->start;
161         for(;;) {
162             if (this)
163                 new->end = this->start;
164             else
165                 new->end = root->end;
166             if (new->start < min)
167                 new->start = min;
168             if (new->end > max)
169                 new->end = max;
170             new->start = (new->start + align - 1) & ~(align - 1);
171             if (alignf)
172                 alignf(alignf_data, new, size);
173             if (new->start < new->end && new->end - new->start + 1 >= size) {
174                 new->end = new->start + size - 1;
175                 return 0;
```

```
176             }
177             if (!this)
178                 break;
179             new->start = this->end + 1;
180             this = this->sibling;
181         }
182         return -EBUSY;
183     }
```

具体的分配并不复杂。如果总线的地址区间是个整体，由 root 所指的 resource 数据结构是个叶节点，那么其起点就用作待分配区间的起点，然后将起点按要求进行边界对齐，并作必要的调整，再来检查区间的大小。否则，要是 root->child 非 0，则需要扫描各个子区间，从中发现足够大的子区间。分配成功时函数返回 0，否则返回－EBUSY。初步确定了分配的起始地址，并且作了边界对齐以后，可能(如果 alignf 非 0)还要调整一下起点地址。在这里，用来调整的函数是 pcibios_align_resource( )，其代码在 arch/i386/kernel/pci-i386.c 中：

[pci_init( ) > pcibios_init( ) > pcibios_resource_survey ( ) > pcibios_assign_resources( )
 > pci_assign_resource( ) > pci_assign_bus_resource( ) > allocate_resource( )
 > find_resource( ) > pcibios_align_resource( )]

```
125     /*
126      * We need to avoid collisions with `mirrored' VGA ports
127      * and other strange ISA hardware, so we always want the
128      * addresses to be allocated in the 0x000-0x0ff region
129      * modulo 0x400.
130      *
131      * Why? Because some silly external IO cards only decode
132      * the low 10 bits of the IO address. The 0x00-0xff region
133      * is reserved for motherboard devices that decode all 16
134      * bits, so it's ok to allocate at, say, 0x2800-0x28ff,
135      * but we want to try to avoid allocating at 0x2900-0x2bff
136      * which might have be mirrored at 0x0100-0x03ff..
137      */
138     void
139     pcibios_align_resource(void *data, struct resource *res, unsigned long size)
140     {
141         if (res->flags & IORESOURCE_IO) {
142             unsigned long start = res->start;
143
144             if (start & 0x300) {
145                 start = (start + 0x3ff) & ~0x3ff;
146                 res->start = start;
147             }
148         }
149     }
```

这种调整只是针对 I/O 地址的(141 行)。为什么要作调整呢？这是因为在早期的 PC 机中规定外设接口卡只能使用 4KB 以下的 I/O 地址，而其中 0x00～0xff 又是保留给母板使用的。所以，早期的接口卡往往只对 I/O 地址的低 10 位，即 4KB 的范围解码，但是同时这些接口卡又不会使用 0x00～0xff，即 bit8 和 bit9 为 0 的这段区间。因此，要避免使用地址中 bit8 或 bit9 为非 0 的那些地址，以免与这样的接口卡冲突。不过，实际上 PCI 设备通常都将寄存器映射到存储器空间(而不是 I/O 空间)，所以真正需要作出这种调整的机会是不多的。

回到 allocate_resource( )的代码中，如果对 find_resource( )的调用成功了，找到了所需的地址资源，就要进一步通过__request_resource( )将代表着新分配区间的 resource 结构 new 插入 root 的 child 队列中。我们已在前面看过这个函数的代码。至此为止，这种“分配”还只是“账面”上的，只是在 resource 结构树中进行，还没有把分配到的地址设置到具体的设备中去。 函数成功时返回 0，失败则返回一个非 0 指针。

为目标设备上的目标区间分配了总线地址以后，便返回到 pci_assign_bus_resource( )中，接着便通过 pcibios_update_resource( )将起始地址设置到目标设备中，从而建立起地址映射。这个函数的代码在 arch/i386/kernel/pci-i386.c 中：

[pci_init( ) > pcibios_init( ) > pcibios_resource_survey ( ) > pcibios_assign_resources( ) > pci_assign_resource( ) > pci_assign_bus_resource( ) > pcibios_update_resource( )]

```
 97    void
 98    pcibios_update_resource(struct pci_dev *dev, struct resource *root,
 99                struct resource *res, int resource)
100    {
101        u32 new, check;
102        int reg;
103
104        new = res->start | (res->flags & PCI_REGION_FLAG_MASK);
105        if (resource < 6) {
106            reg = PCI_BASE_ADDRESS_0 + 4*resource;
107        } else if (resource == PCI_ROM_RESOURCE) {
108            res->flags |= PCI_ROM_ADDRESS_ENABLE;
109            new |= PCI_ROM_ADDRESS_ENABLE;
110            reg = dev->rom_base_reg;
111        } else {
112        /* Somebody might have asked allocation of a non-standard resource */
113            return;
114        }
115
116        pci_write_config_dword(dev, reg, new);
117        pci_read_config_dword(dev, reg, &check);
119            if ((new ^ check) & ((new & PCI_BASE_ADDRESS_SPACE_IO) ?
                    PCI_BASE_ADDRESS_IO_MASK : PCI_BASE_ADDRESS_MEM_MASK)) {
119            printk(KERN_ERR "PCI: Error while updating region "
120                    "%s/%d (%08x != %08x)\n", dev->slot_name, resource,
121                    new, check);
```

```
122         }
123     }
```

这里的参数 res 指向目标设备上目标区间的 resource 数据结构，其中的字段 start 就是为其分配的(起始)总线地址，整数(下标)resource 则表明是设备中的哪一个区间。如前所述，这个总线地址一定是与 16 字节边界对齐的，所以其最低 4 位用于控制目的，这些标志位来自 res->flags 的最低 4 位。总线地址的设置和映射的建立倒是很简单的，只要把所分配的总线地址连同标志位写入目标设备的配置寄存器组中相应区间的寄存器就行了。这里 116 行将新的地址通过配置寄存器组写入目标设备，然后再读回来加以验证。

不过，设置了区间的地址并不意味着这个区间已经可以访问了。配置寄存器组中的命令寄存器里有两个控制位，即 PCI_COMMAND_IO 和 PCI_COMMAND_MEMORY，就好像是两个总开关，分别控制着设备的所有 I/O 地址区间和存储器地址区间。最终，还要把这两个控制位都设成 1 才能使这些区间真正地连接到 PCI 总线上。我们看到，这里并没有走出这最后一步，实际上是留给了具体的设备驱动程序。

完成了对所有设备、所有区间的地址设置以后，逐层返回到 pci_init( )中，还要在一个循环中对每个设备调用 pci_fixup_device( )。我们已在前面看过它的代码，不过上一次调用时的参数 pass 为 PCI_FIXUP_HEADER，进行的是对从设备读出的头部信息的修正；而这一次为 PCI_FIXUP_FINAL，是对设置了地址以后的修正。

至此，对 PCI 总线的初始化已经完成，内存中已经建立起代表着全部 PCI 总线和 PCI 设备的若干棵(通常只是一棵)树，而每项设备的每个地址区间都已有了总线地址。这样，给定一项设备(更确切地说是一项功能)的有关信息，例如设备的类型、由谁制造，就可以通过搜索这些树找到代表着此项设备的 pci_dev 数据结构。为此，内核提供了一个函数 pci_find_device( )，其代码在 drivers/pci/pci.c 中：

```
96     struct pci_dev *
97     pci_find_device(unsigned int vendor, unsigned int device,
                                              const struct pci_dev *from)
98     {
99         return pci_find_subsys(vendor, device, PCI_ANY_ID, PCI_ANY_ID, from);
100    }
```

这个函数的主体是 pci_find_subsys( )，其代码也在同一文件中：

```
63     struct pci_dev *
64     pci_find_subsys(unsigned int vendor, unsigned int device,
65             unsigned int ss_vendor, unsigned int ss_device,
66             const struct pci_dev *from)
67     {
68         struct list_head *n = from ? from->global_list.next : pci_devices.next;
69 
70         while (n != &pci_devices) {
71             struct pci_dev *dev = pci_dev_g(n);
72             if ((vendor == PCI_ANY_ID || dev->vendor == vendor) &&
```

```
73                  (device == PCI_ANY_ID || dev->device == device) &&
74              (ss_vendor == PCI_ANY_ID || dev->subsystem_vendor == ss_vendor) &&
75              (ss_device == PCI_ANY_ID || dev->subsystem_device == ss_device))
76                  return dev;
77              n = n->next;
78          }
79          return NULL;
80      }
```

当 PCI 总线上有多个来自同一厂商的同种设备时(例如两块 Ethernet 接口卡)，可以逐次以不同的起点 from 调用 pci_find_device( )，直至找到所有这些设备的 pci_dev 数据结构。

如果关心的不是由谁制造，而是模块的功能和用途，则可以通过 pci_find_class( )寻找，其代码也在 drivers/pci/pci.c 中：

```
115     struct pci_dev *
116     pci_find_class(unsigned int class, const struct pci_dev *from)
117     {
118         struct list_head *n = from ? from->global_list.next : pci_devices.next;
119
120         while (n != &pci_devices) {
121             struct pci_dev *dev = pci_dev_g(n);
122             if (dev->class == class)
123                 return dev;
124             n = n->next;
125         }
126         return NULL;
127     }
```

找到了目标设备的 pci_dev 数据结构，就可以通过其指针数组 resource[ ]找到各个总线地址区间的 resource 结构，从而取得各个区间的总线地址。在这个基础上，通过__ioremap( )为这些区间建立起虚拟地址的映射，再通过设备的命令寄存器打开其所有区间，就可以像访问内存空间一样地访问这些区间了。此外，如果目标设备具有中断功能，则还要进一步落实好中断请求线。内核为设备驱动程序提供了一个 inline 函数 pcibios_enable_device( )来帮助驱动程序做这些事情，其代码在 include/linux/pci.h 中：

```
1042    int pcibios_enable_device(struct pci_dev *dev)
1043    {
1044        int err;
1045
1046        if ((err = pcibios_enable_resources(dev)) < 0)
1047            return err;
1048        pcibios_enable_irq(dev);
1049        return 0;
1050    }
```

首先就是通过 pcibios_enable_resources( ) 打开设备的各个地址区间，其代码在 arch/i386/kernel/pci-i386.c 中：

[pcibios_enable_device( ) > pcibios_enable_resources( )]

```
306   int pcibios_enable_resources(struct pci_dev *dev)
307   {
308       u16 cmd, old_cmd;
309       int idx;
310       struct resource *r;
311
312       pci_read_config_word(dev, PCI_COMMAND, &cmd);
313       old_cmd = cmd;
314       for(idx=0; idx<6; idx++) {
315           r = &dev->resource[idx];
316           if (!r->start && r->end) {
317               printk(KERN_ERR "PCI: Device %s not available because of resource
                  collisions\n", dev->slot_name);
318               return -EINVAL;
319           }
320           if (r->flags & IORESOURCE_IO)
321               cmd |= PCI_COMMAND_IO;
322           if (r->flags & IORESOURCE_MEM)
323               cmd |= PCI_COMMAND_MEMORY;
324       }
325       if (dev->resource[PCI_ROM_RESOURCE].start)
326           cmd |= PCI_COMMAND_MEMORY;
327       if (cmd != old_cmd) {
328           printk("PCI: Enabling device %s (%04x -> %04x)\n",
                                      dev->slot_name, old_cmd, cmd);
329           pci_write_config_word(dev, PCI_COMMAND, cmd);
330       }
331       return 0;
332   }
```

这段代码很简单，我们就不多加解释了。

接着是对中断请求线的处理，目的是要搞清楚该设备的中断请求最终连接到了中断控制器的哪一条输入线，这样才能正确地登记其中断服务程序。函数 pcibios_enable_irq( ) 的代码在 arch/i386/kernel/pci-irq.c 中：

[pcibios_enable_device( ) > pcibios_enable_resources( )]

```
613   void pcibios_enable_irq(struct pci_dev *dev)
614   {
615       u8 pin;
616       pci_read_config_byte(dev, PCI_INTERRUPT_PIN, &pin);
617       if (pin && !pcibios_lookup_irq(dev, 1) && !dev->irq) {
```

```
618            char *msg;
619            if (io_apic_assign_pci_irqs)
620                msg = " Probably buggy MP table.";
621            else if (pci_probe & PCI_BIOS_IRQ_SCAN)
622                msg = "";
623            else
624                msg = " Please try using pci=biosirq.";
625            printk(KERN_WARNING
                   "PCI: No IRQ known for interrupt pin %c of device %s.%s\n",
626                   'A' + pin - 1, dev->slot_name, msg);
627        }
628    }
```

这里的 617 行表示：如果从寄存器 PCI_INTERRUPT_PIN 读出的数值非 0，就说明设备具有中断功能，因此调用 pcibios_lookup_irq( )寻找其中断请求线的去向。要是这个函数正常返回，但 dev->irq 仍为 0，那就说明有问题了。

前面，在 pcibios_fixup_irqs( )中也曾对所有设备逐个地调用过 pcibios_lookup_irq( )，当时的第二个调用参数为 0，表示如果其中断请求线尚未连接就留着再说。而现在，则第二个调用参数为 1，表示若尚未连接就要选择一个合适的对象并完成连接。为什么要把这一步留到现在呢？一方面这是“lazy computaion”，插在总线上的设备未必就是实际要用的，如果花费了代价而实际上并不使用是一种浪费，所以留到具体设备(也许是可安装模块)初始化的时候再来处理是合理的。另一方面，更为重要的是，当时在循环尚未结束之前并不知道系统中到底有多少设备连在同一条 PCI 中断请求线上，也不清楚中断控制器的各条输入线的负荷，因而不易作出真正合理的选择；而现在，则显然可以作出更合理的选择了。我们再看一下 pcibios_lookup_irq( )有关的两个片段(arch/i386/kernel/pci-irq.c)：

[pcibios_enable_device( ) > pcibios_enable_resources( ) > pcibios_lookup_irq( )]

```
442        /*
443         * Find the best IRQ to assign: use the one
444         * reported by the device if possible.
445         */
446        newirq = dev->irq;
447        if (!newirq && assign) {
448            for (i = 0; i < 16; i++) {
449                if (!(mask & (1 << i)))
450                    continue;
451                if (pirq_penalty[i] < pirq_penalty[newirq] &&
452    !request_irq(i, pcibios_test_irq_handler, SA_SHIRQ, "pci-test", dev)) {
453                    free_irq(i, dev);
454                    newirq = i;
455                }
456            }
457        }
458        DBG(" -> newirq=%d", newirq);
```

```
459
        . . . . .

477         } else if (newirq && r->set && (dev->class >> 8) != PCI_CLASS_DISPLAY_VGA) {
478             DBG(" -> assigning IRQ %d", newirq);
479             if (r->set(pirq_router_dev, dev, pirq, newirq)) {
480                 eisa_set_level_irq(newirq);
481                 DBG(" ... OK\n");
482                 msg = "Assigned";
483                 irq = newirq;
484             }
485         }
```

这两个片断的逻辑其实都是很简单的，读者在前面读过这个函数的其余部分，所以从总体上不应该有困难。但是，对 452 行却需要作一点说明。对于中断控制器一侧候选的中断请求输入线，这里通过 request_irq( )（见第 3 章）试着登记一个空函数 pcibios_test_irq_handler( )，如果成功再通过 free_irq( ) 将其撤销，这样就保证了以后登记真正的中断服务程序时也能成功。否则，如果试登记失败，就不能把这条输入线作为候选。代码中还调用了一个函数 eisa_set_level_irq( )，那完全是因为硬件的特殊要求，我们就不关心了。

有些设备的情况还要更复杂，所以内核中又提供了更为一般化的 inline 函数 pci_module_init( )（以及与之有关的一系列函数）来帮助设备驱动程序的设计和实现，读者可参阅本章第 9 节“通用串行外部总线 USB”。

8.5 块设备的驱动

块设备是文件系统的物质基础。而文件系统，则就是对块设备上所存储的内容按某种格式加以组织的结果。因此，对块设备（内容）的访问就有两种方式。一种是忽略对其内容的组织，即忽略文件系统的存在，而将其看成一个记录块的阵列（数组）；另一种则遵循文件系统的组织，将其看成通过各层目录组织在一起的文件集合，而每个文件则又是若干记录块的有序集合。由于“记录块数组”和“记录块的有序集合”都是有序的，在逻辑上又可以把它们看成线性的“字节流”。以书本为例，我们可以把一本书看成一叠页面，按页号找到其中的内容；也可以把它看成由若干章节构成，而按第几章第几节找到其内容。同时，我们可以说第几页上的第几个字是什么，乃至整本书（假定每页上的字数是固定的）的第几个字是什么；也可以说第几章第几节的第几个字是什么。当打开一个代表块设备的文件节点，对这个“块设备文件”读/写时，我们把这设备(的内容)看作一个字节流，而忽略对其内容的组织即文件系统的存在。同样，当打开一个存在于块设备上的普通文件时，我们把这个特定的文件看作一个字节流，但是这个字节流按一定的格式和规则映射到块设备上。前者在 Unix 中称为“原始设备”，并且看成是字符设备，以示与后者的区别。这一点在 Linux 中有了改变，块设备还是块设备，只要是通过设备文件（节点）访问就是“原始”的。而在安装以后的块设备上，对普通文件的访问则自然不能忽略文件系统的组织了。这里还要说明，即使是对“原始”设备的访问，首先也必须在文件系统中

找到代表着这个设备的文件节点，在这个过程中当然不能忽略文件系统的组织。

在"文件的读与写"一节中，读者已经看到对普通文件的访问怎样变换成了对块设备上记录块的访问，最后转化成对函数 ll_rw_block( )的调用，从这个函数开始就进入了设备层。我们在这一节中继续往下看，不过在此之前还要先看一下对"原始"块设备文件的访问，包括打开文件以及对文件的读/写。从概念上说，这些内容或许应该放在"文件系统"一章中，但是因为设备文件的"文件层"很薄，其内容又与设备驱动关系很紧密，所以我们把它放在这里与设备驱动一起介绍。

先看块设备文件的打开。在"系统调用 mknod( )"一节中读者已经看到，设备文件是通过 mknod( )创建的，创建时将设备的类型（块设备或字符设备）、主设备号、次设备号都写入了代表该文件的索引节点中。这个索引节点本身也存在于一个块设备上，具体取决于该设备文件节点所在的文件系统。同时它又代表着一个块设备。二者可以相同，也可以不同。打开文件时，在 path_walk( )中找到该设备文件节点所在的目录，再根据目录项的指引从文件系统所在的设备上读入这个索引节点，并检验它的模式，看看它所代表的是普通文件，还是目录、符号连接，或者是包括设备文件在内的特殊文件。就 Ext2 文件系统而言，这部分操作是在函数 ext2_read_inode( )中进行的。我们已经在"从路径名到目标节点"一节中列出了这个函数的代码，读者可以回过去重读一下。对于包括设备文件在内的特殊文件，这个函数调用 init_sperial_inode( )来初始化为这个文件创建的 inode 数据结构，这是在 fs/ext2/inode.c 中的第 1077 行执行的：

```
961     void ext2_read_inode (struct inode * inode)
962     {
        . . . . . .
1077            init_special_inode(inode, inode->i_mode,
1078                        le32_to_cpu(raw_inode->i_block[0]));
        . . . . . .
1102    }
```

这里 inode 指向为该文件创建的 inode 结构，它的 i_mode 字段已经根据从设备上读入的索引节点中的有关信息设置好了，而 raw_inode 则指向从设备上读入的索引节点，即 ext2_inode 数据结构。我们已经在"系统调用 mknod( )"一节中读过 init_special_inode( )的代码，此处再列出其中有关块设备的几行（fs/devices）：

```
200     void init_special_inode(struct inode *inode, umode_t mode, int rdev)
201     {
        . . . . . .
206         } else if (S_ISBLK(mode)) {
207             inode->i_fop = &def_blk_fops;
208             inode->i_rdev = to_kdev_t(rdev);
209             inode->i_bdev = bdget(rdev);
210         } else if (S_ISFIFO(mode))
            . . . . . .
216             }
```

经过了这几行，inode 结构中的指针 i_fop 就指向了具体块设备的 file_operations 数据结构。同时，

inode 结构中还有一个专用于块设备的指针 i_bdev，指向代表着具体块设备的 block_device 数据结构。这个数据结构或许已经存在于内存中，或许需要创建，由设备号惟一地加以确定。在“系统调用 mknod( )”一节中已经列出了函数 bdget( )的代码，块设备的 file_operations 数据结构定义见 fs/block_dev.c：

```
709	struct file_operations def_blk_fops = {
710		open:		blkdev_open,
711		release:	blkdev_close,
712		llseek:		block_llseek,
713		read:		block_read,
714		write:		block_write,
715		fsync:		block_fsync,
716		ioctl:		blkdev_ioctl,
717	};
```

这样，就为对 blkdev_open( )的调用铺平了道路。当 path_walk( )结束的时候，目标文件的 inode 结构已经与块设备的文件层挂上了钩。不过此时具体 block_device 结构中的指针 bd_op 还没有设置（如果是新创建的 block_device 结构），也就是还没有与具体块设备的操作挂上钩，那要到进入 blkdev_open( )以后再来处理。所以这也是个走一步看一步，“摸着石头过河”的过程。

最后，在 dentr_open( )中（见“文件的打开与关闭”）把目标文件 inode 结构中的指针 f_ops 复制到 file 结构中（f_op 指针），并通过相应 file_operations 结构中的函数指针 open 调用 blkdev_open( )。这个函数的代码也在 fs/block_dev.c 中：

```
[sys_open( ) > filp_open( ) > dentry_open( ) > blkdev_open( )]

644	int blkdev_open(struct inode * inode, struct file * filp)
645	{
646		int ret = -ENXIO;
647		struct block_device *bdev = inode->i_bdev;
648		down(&bdev->bd_sem);
649		lock_kernel( );
650		if (!bdev->bd_op)
651			bdev->bd_op = get_blkfops(MAJOR(inode->i_rdev));
652		if (bdev->bd_op) {
653			ret = 0;
654			if (bdev->bd_op->open)
655				ret = bdev->bd_op->open(inode, filp);
656			if (!ret)
657				atomic_inc(&bdev->bd_openers);
658			else if (!atomic_read(&bdev->bd_openers))
659				bdev->bd_op = NULL;
660		}
661		unlock_kernel( );
662		up(&bdev->bd_sem);
663		return ret;
```

```
664     }
```

每种具体的块设备都各有一套具体的操作，因而各自有一个类似于 file_operations 那样的数据结构，称为 block_device_operations，其定义见 include/linux/fs.h：

```
760     struct block_device_operations {
761         int (*open) (struct inode *, struct file *);
762         int (*release) (struct inode *, struct file *);
763         int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
764         int (*check_media_change) (kdev_t);
765         int (*revalidate) (kdev_t);
766     };
```

如果说 file_operations 数据结构是连接虚拟的、抽象的 vfs 文件操作与具体文件系统（或文件类型）的文件操作之间的枢纽，那么 block_device_operations 就是连接抽象的块设备操作与具体块设备类型的操作之间的枢纽。块设备的类型是由主设备号惟一确定的，所以主设备号惟一地确定了一个具体的 block_device_operations 数据结构，而 blkdev_open( )的任务就是根据主设备号找到相应的数据结构，并使 block_device 结构中的指针指向这个数据结构，然后调用由这个数据结构中的函数指针 open 所指向的函数（如果该指针不是 NULL）。如果某种设备在打开它的时候不需要做任何事，那么它的 block_device_operations 结构中的指针 open 就是 NULL。

寻找具体块设备类型的 block_device_operations 数据结构是由 get_blkfops( )完成的，其代码在 fs/block_dev.c 中：

[sys_open( ) > filp_open( ) > dentry_open( ) > blkdev_open( ) > get_blkfops( )]

```
487     /*
488         Return the function table of a device.
489         Load the driver if needed.
490     */
491     const struct block_device_operations * get_blkfops(unsigned int major)
492     {
493         const struct block_device_operations *ret = NULL;
494 
495         /* major 0 is used for non-device mounts */
496         if (major && major < MAX_BLKDEV) {
497     #ifdef CONFIG_KMOD
498             if (!blkdevs[major].bdops) {
499                 char name[20];
500                 sprintf(name, "block-major-%d", major);
501                 request_module(name);
502             }
503     #endif
504             ret = blkdevs[major].bdops;
505         }
506         return ret;
```

```
507     }
```

内核中有一个数组 blkdevs[ ]，以主设备号为下标就可以通过数组中的表项找到各种设备的 block_device_operations 数据结构。这个数组也是在 fs/block_dev.c 中定义的：

```
468     static struct {
469         const char *name;
470         struct block_device_operations *bdops;
471     } blkdevs[MAX_BLKDEV];
```

内核在初始化时根据系统的配置将有关块设备的block_device_operations结构连同设备名一起登记入数组中的相应表项。这样，get_blkfops( )只要以主设备号为下标就可以找到给定设备的这个数据结构了（见代码中的第 504 行）。值得注意的是，如果内核支持可安装模块，那么即使内核在初始化时没有登记某种块设备，也可以在需要时通过 reguest_module( )把用来支持该种设备的代码和数据结构安装进来。对这种模块的命名有个规则，那就是"block-major-"加上具体设备的主设备号（见代码 500 行）。

在这里我们假定所用的块设备为 IDE 硬盘，因为这是最常用的。在 PC 机上最多可以有主、次（primary/secondary）两个 IDE 接口，每个 IDE 接口又可以支持主、从（master/slave）共两个 IDE 硬盘，所以最多可以有 4 个 IDE 硬盘（包括光盘），其中第一个 IDE 接口上主硬盘的主设备号为 3（其余硬盘的主设备号依次为 22、33 和 34）。IDE 硬盘都带有内装的控制器(IDE 为 Integrated Drives Electronics 的缩写)，所以只需要"接口"而并不需要"控制卡"。在过去的十年中，这种硬盘有了很大的发展，以至于早期 IDE 硬盘所用的 block_device_operations 数据结构 hd_ops 已经不适用于新型的 IDE 硬盘，而只好另外定义新的数据结构 ide_fops，其定义在 drivers/ide/ide.c 中：

```
3492    struct block_device_operations ide_fops[ ] = {{
3493        open:           ide_open,
3494        release:        ide_release,
3495        ioctl:          ide_ioctl,
3496        check_media_change: ide_check_media_change,
3497        revalidate:     ide_revalidate_disk
3498    }};
```

注意，这里说的是定义新的数据结构，而不是数据结构类型.之所以定义新的数据结构是因为有了一组新的函数。实际上，这里定义的是一个结构数组，只是数组中只有一个表项。这样就为将来预留下进一步的发展空间。

回到 blkdev_open( )的代码中。至此，我们已经把对抽象的"块设备文件"的操作逐层地具体化为对块设备的操作。但是"块设备"还是个抽象的概念，还需要进一步具体化，这就需要进入为 IDE 硬盘提供的操作了。现在，block_device 结构中的指针 bd_op 已经指向新型 IDE 硬盘的 block_device_operations 结构 ide_fops，而其中的指针 open 又指向 ide_open( )，所以函数 blkdev_open( ) 代码中的 655 行就通过这个指针调用 ide_open( )，其代码在 drivers/ide/ide.c 中：

[sys_open( ) > filp_open( ) > dentry_open( ) > blkdev_open( ) > ide_open( )]

```
1832    static int ide_open (struct inode * inode, struct file * filp)
1833    {
1834        ide_drive_t *drive;
1835        int rc;
1836
1837        if ((drive = get_info_ptr(inode->i_rdev)) == NULL)
1838            return -ENXIO;
1839        MOD_INC_USE_COUNT;
1840        if (drive->driver == NULL)
1841            ide_driver_module( );
1842    #ifdef CONFIG_KMOD
1843        if (drive->driver == NULL) {
1844            if (drive->media == ide_disk)
1845                (void) request_module("ide-disk");
1846            if (drive->media == ide_cdrom)
1847                (void) request_module("ide-cd");
1848            if (drive->media == ide_tape)
1849                (void) request_module("ide-tape");
1850            if (drive->media == ide_floppy)
1851                (void) request_module("ide-floppy");
1852        }
1853    #endif /* CONFIG_KMOD */
1854        while (drive->busy)
1855            sleep_on(&drive->wqueue);
1856        drive->usage++;
1857        if (drive->driver != NULL) {
1858            if ((rc = DRIVER(drive)->open(inode, filp, drive)))
1859                MOD_DEC_USE_COUNT;
1860            return rc;
1861        }
1862        printk ("%s: driver not present\n", drive->name);
1863        drive->usage--;
1864        MOD_DEC_USE_COUNT;
1865        return -ENXIO;
1866    }
```

数据结构 ide_ops[ ]中提供了对 IDE 硬盘操作的函数跳转表，但是对具体 IDE 硬盘的操作还需要有关具体设备的许多数据，需要有个具体设备的“控制块”，这就是 ide_drive_t 数据结构，有关的定义在 include/linux/ide.h 中：

```
257     /*
258      * Now for the data we need to maintain per-drive:  ide_drive_t
259      */
260
261     #define ide_scsi      0x21
262     #define ide_disk      0x20
263     #define ide_optical 0x7
```

```
264  #define ide_cdrom   0x5
265  #define ide_tape    0x1
266  #define ide_floppy  0x0
267
268  typedef union {
269      unsigned all             : 8;      /* all of the bits together */
270      struct {
271          unsigned set_geometry   : 1;   /* respecify drive geometry */
272          unsigned recalibrate    : 1;   /* seek to cyl 0      */
273          unsigned set_multmode   : 1;   /* set multmode count */
274          unsigned set_tune    : 1;      /* tune interface for drive */
275          unsigned reserved    : 4;      /* unused */
276      } b;
277  } special_t;
278
279  typedef struct ide_drive_s {
280      request_queue_t      queue;      /* request queue */
281      struct ide_drive_s  *next;        circular list of hwgroup drives */
282      unsigned long sleep;             /* sleep until this time */
283      unsigned long service_start;     /* time we started last request */
284      unsigned long service_time;      /* service time of last request */
285      unsigned long timeout;           /* max time to wait for irq */
286      special_t   special;             /* special action flags */
287      byte     keep_settings;          /* restore settings after drive reset */
288      byte     using_dma;              /* disk is using dma for read/write */
289      byte     waiting_for_dma;        /* dma currently in progress */
290      byte     unmask;                 /* flag: okay to unmask other irqs */
291      byte     slow;                   /* flag: slow data port */
292      byte     bswap;                  /* flag: byte swap data */
293      byte     dsc_overlap;            /* flag: DSC overlap */
294      byte     nice1;                  /* flag: give potential excess bandwidth */
295      unsigned present     : 1;        /* drive is physically present */
296      unsigned noprobe     : 1;        /* from:  hdx=noprobe */
297      unsigned busy        : 1;        /* currently doing revalidate_disk( ) */
298      unsigned removable   : 1;        /* 1 if need to do check_media_change */
299      unsigned forced_geom     : 1;    /* 1 if hdx=c,h,s was given at boot */
300      unsigned no_unmask   : 1;        /* disallow setting unmask bit */
301      unsigned no_io_32bit     : 1;    /* disallow enabling 32bit I/O */
302      unsigned nobios      : 1;        /* flag: do not probe bios for drive */
303      unsigned revalidate : 1;         /* request revalidation */
304      unsigned atapi_overlap: 1;       /* flag: ATAPI overlap (not supported) */
305      unsigned nice0   : 1;            /* flag: give obvious excess bandwidth */
306      unsigned nice2   : 1;            /* flag: give a share in our own bandwidth */
307      unsigned doorlocking: 1;         /* flag: for removable only: door
                                                  lock/unlock works */
308      unsigned autotune    : 2;        /* 1=autotune, 2=noautotune, 0=default */
309      unsigned remap_0_to_1: 2;        /* 0=remap if ezdrive, 1=remap, 2=noremap */
310      unsigned ata_flash   : 1;        /* 1=present, 0=default */
```

```
311         byte        scsi;             /* 0=default, 1=skip current ide-subdriver
                                                    for ide-scsi emulation */
312         byte        media;            /* disk, cdrom, tape, floppy, ... */
313         select_t    select;           /* basic drive/head select reg value */
314         byte        ctl;              /* "normal" value for IDE_CONTROL_REG */
315         byte        ready_stat;       /* min status value for drive ready */
316         byte        mult_count;       /* current multiple sector setting */
317         byte        mult_req;         /* requested multiple sector setting */
318         byte        tune_req;         /* requested drive tuning setting */
319         byte        io_32bit;         /* 0=16 bit, 1=32-bit, 2/3=32bit+sync */
320         byte        bad_wstat;        /* used for ignoring WRERR_STAT */
321         byte        nowerr;           /* used for ignoring WRERR_STAT */
322         byte        sect0;            /* offset of first sector for DM6:DDO */
323         byte        usage;            /* current "open( )" count for drive */
324         byte        head;             /* "real" number of heads */
325         byte        sect;             /* "real" sectors per track */
326         byte        bios_head;        /* BIOS/fdisk/LILO number of heads */
327         byte        bios_sect;        /* BIOS/fdisk/LILO sectors per track */
328         unsigned int    bios_cyl;     /* BIOS/fdisk/LILO number of cyls */
329         unsigned int    cyl;          /* "real" number of cyls */
330         unsigned long   capacity;     /* total number of sectors */
331         unsigned int drive_data;      /* for use by tuneproc/selectproc as needed */
332         void            *hwif;        /* actually (ide hwif t *) */
333         wait_queue_head_t wqueue;     /* used to wait for drive in open( ) */
334         struct hd_driveid *id;        /* drive model identification info */
335         struct hd_struct  *part;      /* drive partition table */
336         char        name[4];          /* drive name, such as "hda" */
337         void        *driver;          /* (ide_driver_t *) */
338         void        *driver_data;     /* extra driver data */
339         devfs_handle_t  de;           /* directory for device */
340         struct proc_dir_entry *proc;  /* /proc/ide/ directory entry */
341         void        *settings;        /* /proc/ide/ drive settings */
342         char        driver_req[10];   /* requests specific driver */
343         int     last_lun;             /* last logical unit */
344         int     forced_lun;           /* if hdxlun was given at boot */
345         int     lun;                  /* logical unit */
346         int   crc_count;              /* crc counter to reduce drive speed */
347         byte  quirk_list;   /* drive is considered quirky if set for a specific host */
348         byte  suspend_reset;    /* drive suspend mode flag, soft-reset recovers */
349         byte        init_speed;       /* transfer rate set at boot */
350         byte        current_speed;    /* current transfer rate set */
351         byte        dn;               /* now wide spread use */
352     } ide_drive_t;
```

代码作者对结构中各个字段的意义和作用已经加了注释，以后随着代码的进展还会变得更加清楚。每个具体的 IDE 磁盘都有这么一个数据结构，函数 get_info_ptr( )根据设备号找到这个数据结构并返回指向它的指针，这个函数的代码在 drivers/ide/ide.c 中：

[sys_open( ) > filp_open( ) > dentry_open( ) > blkdev_open( ) > ide_open( ) > get_info_ptr( )]

```
1628    /*
1629     * get info ptr( ) returns the (ide_drive_t *) for a given device number.
1630     * It returns NULL if the given device number does not match any present
            drives.
1631     */
1632    ide_drive_t *get_info_ptr (kdev_t i_rdev)
1633    {
1634        int     major = MAJOR(i_rdev);
1635    #if 0
1636        int     minor = MINOR(i rdev) & PARTN_MASK;
1637    #endif
1638        unsigned int    h;
1639
1640        for (h = 0; h < MAX_HWIFS; ++h) {
1641            ide_hwif_t  *hwif = &ide_hwifs[h];
1642            if (hwif->present && major == hwif->major) {
1643                unsigned unit = DEVICE NR(i_rdev);
1644                if (unit < MAX_DRIVES) {
1645                    ide_drive_t *drive = &hwif->drives[unit];
1646    #if 0
1647                    if ((drive->present) && (drive->part[minor].nr sects))
1648    #else
1649                    if (drive->present)
1650    #endif
1651                        return drive;
1652                }
1653                break;
1654            }
1655        }
1656        return NULL;
1657    }
```

内核中有个数组 ide_hwifs[ ]，数组的每个元素都是一个 ide_hwif_t 数据结构，代表着系统中的一个可能的 IDE 接口（本节后面会给出这个数据结构的定义）。系统初始化时如果检测到一个 IDE 接口，就把相应表项中的 present 字段设置成 1。同时，这种数据结构中又有个 ide_drive_t 结构数组 drives[ ]。初始化时如果检测到某个接口上有磁盘相连，就将相应 ide_drive_t 结构中的 present 字段也设成 1，并根据检测到或从系统的 CMOS 芯片中读到的各项参数设置这个数据结构。也就是说，ide_hwif_t 数据结构是对 IDE 接口的描述，而 ide_drive_t 数据结构是对连接在具体 IDE 接口上的“IDE 设备”的描述。例如，如果在系统的主（primary）IDE 接口上检测到有主/从两个磁盘相连，就把这两个磁盘的参数分别填入 ide_hwifs[0]中的 drives[0]和 drives[1]，并把它们的 present 字段设置成 1。再例如，如果在次(secondary)IDE 接口上连接着一个 Mitsumi CDROM，那就把它的参数填入 ide_hwifs[1]里面的 drives[0]，并且把 ide_hwifs[1]中的字段 major 设置成 MITSUMI_CDROM_MAJOR。然后，当需要时，就由 get_info_ptr( )根据主设备号在 ide_hwifs[ ]中搜索，找到相应的接口后再根据次设备号找到连接在该接

口上的具体磁盘的 ide_drive_t 数据结构。

如果 get_info_ptr( )找不到所要求的 ide_drive_t 结构，就说明系统中不存在相应的硬盘，所以 ide_open( )返回出错代码－ENXIO，整个 open( )操作也就失败了。

在 ide_drive_t 结构中(337 行)有个 void 指针 driver，可以根据不同的要求指向不同的 ide_driver_t 数据结构（注意 ide_driver_t 和 ide_drive_t 是两种不同的数据结构）。这个指针在系统初始化过程中检测到 IDE 接口上的设备时，根据设备的类型而设置成指向不同类型的数据结构。对于 IDE 硬盘，它指向一个 ide_driver_t 数据结构 idedisk_driver。同类的数据结构还有 idetape_driver、ide_cdrom_driver 以及 ide_floppy_driver，分别代表着可以连接到 IDE 接口上的不同类型的设备。

如果通过 get_info_ptr( )找到了所需的 ide_drive_t 数据结构，但是它的 driver 指针却是 NULL，就说明初始化时虽然检测到了硬盘的存在，但是却因某种原因而未能完成对设备以及数据结构的初始化，所以在 ide_open( )中调用 ide_driver_module( )再试一次。

另一方面，如果内核支持可安装模块，那也可能是因为支持具体 IDE 设备的模块尚未安装或已被拆除而引起，所以试图根据具体的设备类型装入有关的模块。由 ide_driver_module( )启动的操作是异步的，当前进程睡眠等待对设备的操作完成(1854～1855 行)以后再检查 ide_drive_t 结构中的指针 driver。如果已经变成非 0，就表示该设备仍旧与系统相连，并且已经成功地完成了初始化，所以可以通过 ide_driver_t 结构中的函数指针 open 启动特定设备的打开文件操作了。否则，要是 ide_drive_t 结构中的指针 driver 仍是 NULL，那就无法继续而失败了。

如上所述，IDE 硬盘的 ide_driver_t 数据结构是 idedisk_driver，其定义见 ide_disk.c：

```
711     /*
712      *  IDE subdriver functions, registered with ide.c
713      */
714     static ide_driver_t idedisk_driver = {
715         "ide-disk",             /* name */
716         IDEDISK_VERSION,        /* version */
717         ide_disk,               /* media */
718         0,                      /* busy */
719         1,                      /* supports_dma */
720         0,                      /* supports_dsc_overlap */
721         NULL,                   /* cleanup */
722         do_rw_disk,             /* do_request */
723         NULL,                   /* end_request */
724         NULL,                   /* ioctl */
725         idedisk_open,           /* open */
726         idedisk_release,        /* release */
727         idedisk_media_change,       /* media_change */
728         idedisk_revalidate,     /* revalidate */
729         idedisk_pre_reset,      /* pre_reset */
730         idedisk_capacity,       /* capacity */
731         idedisk_special,        /* special */
732         idedisk_proc            /* proc */
733     };
```

所以通过其指针 open 调用的函数(1858 行)为 idedisk_open( )，其代码也在 ide_disk.c 中：

[sys_open( ) > filp_open( ) > dentry_open( ) > blkdev_open( ) > ide_open( ) > idedisk_open( )]

```
478    static int idedisk_open (struct inode *inode, struct file *filp, ide_drive_t *drive)
479    {
480        MOD_INC_USE_COUNT;
481        if (drive->removable && drive->usage == 1) {
482            check_disk_change(inode->i_rdev);
483            /*
484             * Ignore the return code from door_lock,
485             * since the open( ) has already succeeded,
486             * and the door_lock is irrelevant at this point.
487             */
488            if (drive->doorlocking &&
                       ide_wait_cmd(drive, WIN_DOORLOCK, 0, 0, 0, NULL))
489                drive->doorlocking = 0;
490        }
491        return 0;
492    }
```

大部分 IDE 硬盘的介质都不是可拆卸的，所以这个函数在多数情况下并没有做什么事情。但是这并不意味着从总体上、从结构上说这一步是多余的。例如，对于同属 IDE 设备的 CDROM，相应的函数为 ide_cdrom_open( )，那就是实实在在有事情要做的。

从 ide_open( )正常返回，并且逐层返回到 dentry_open( )中，整个打开块设备文件的操作就基本完成了。回顾一下这整个过程，就是从抽象的 vfs 层文件出发，逐层加以具体化，找到相应的数据结构并把这些数据结构联系在一起，层层打通关节的过程。

(1) file_operations 结构使 vfs 具体化成了特定的文件系统或文件类型（块设备文件）。

(2) block_device 数据结构使代表着抽象意义上的文件的 inode 结构具体化成了“块设备”。

(3) block_device_operations 结构使“块设备文件”操作进一步具体化成了“IDE 设备”操作。

(4) ide_drive_t 结构将笼统的“IDE 设备”具体化成了特定种类的 IDE 设备。

(5) ide_driver_t 结构将某种 IDE 设备的操作具体化成对特定 IDE 硬盘的操作。

我们以前讲过，打开文件的实质就是使一个进程与文件所代表的对象建立起连接。上述逐层具体化的过程实际上就是逐站选择前进方向，直至打通到达目标的道路。这个过程一经完成，就为进一步的文件操作做好了准备。读者也许会问，对普通文件的访问最终也要落实到具体的块设备上，但是为什么在那里就不需要打开具体的块设备文件呢？事实上，当把一个块设备安装到文件系统中去时，也已经走过了类似的路程，在文件系统中的安装点与具体块设备之间建立起了连接，只不过这种连接并不是建立在某个特定的进程与设备之间，并没有为之创建起 file 结构即文件操作的上下文而已。读者如果回过去看一下上册第 5 章“文件系统的安装与拆卸”一节，就会发现有些内容已经在那里提到过，只不过那时候我们的注意力不在块设备的细节，因而没有那么深入。

现在文件已经打开，我们以系统调用 read( )为例来看看对块设备的文件操作。我们知道，对于已经打开的块设备文件，其 file 结构中的 file_operations 结构指针指向 def_blk_fops，而从这个结构中可以

发现其函数指针 read 指向 block_read( )，其代码在 fs/block_dev.c 中：

[sys_read( ) > block_read( )]

```
166     ssize_t block_read(struct file * filp, char * buf, size_t count,
                                                        loff_t *ppos)
167     {
168         struct inode * inode = filp->f_dentry->d_inode;
169         size_t block;
170         loff_t offset;
171         ssize_t blocksize;
172         ssize_t blocksize bits, i;
173         size_t blocks, rblocks, left;
174         int bhrequest, uptodate;
175         struct buffer_head ** bhb, ** bhe;
176         struct buffer_head * buflist[NBUF];
177         struct buffer_head * bhreq[NBUF];
178         unsigned int chars;
179         loff_t size;
180         kdev_t dev;
181         ssize_t read;
182
183         dev = inode->i_rdev;
184         blocksize = BLOCK_SIZE;
185         if (blksize_size[MAJOR(dev)] && blksize_size[MAJOR(dev)][MINOR(dev)])
186             blocksize = blksize_size[MAJOR(dev)][MINOR(dev)];
187         i = blocksize;
188         blocksize_bits = 0;
189         while (i != 1) {
190             blocksize_bits++;
191             i >>= 1;
192         }
193
194         offset = *ppos;
195         if (blk_size[MAJOR(dev)])
196             size=(loff_t)blk_size[MAJOR(dev)][MINOR(dev)]<<BLOCK_SIZE_BITS;
197         else
198             size = (loff_t) INT_MAX << BLOCK_SIZE_BITS;
199
200         if (offset > size)
201             left = 0;
202         /* size - offset might not fit into left, so check explicitly. */
203         else if (size - offset > INT_MAX)
204             left = INT_MAX;
205         else
206             left = size - offset;
207         if (left > count)
208             left = count;
```

```
209         if (left <= 0)
210             return 0;
211         read = 0;
212         block = offset >> blocksize_bits;
213         offset &= blocksize-1;
214         size >>= blocksize_bits;
215         rblocks = blocks = (left + offset + blocksize - 1) >> blocksize_bits;
216         bhb = bhe = buflist;
217         if (filp->f_reada) {
218                 if (blocks < read_ahead[MAJOR(dev)] / (blocksize >> 9))
219                 blocks = read_ahead[MAJOR(dev)] / (blocksize >> 9);
220             if (rblocks > blocks)
221                 blocks = rblocks;
222 
223         }
224         if (block + blocks > size) {
225             blocks = size - block;
226             if (blocks == 0)
227                 return 0;
228         }
229 
230         /* We do this in a two stage process.  We first try to request
231            as many blocks as we can, then we wait for the first one to
232            complete, and then we try to wrap up as many as are actually
233            done.  This routine is rather generic, in that it can be used
234            in a filesystem by substituting the appropriate function in
235            for getblk.
236 
237            This routine is optimized to make maximum use of the various
238            buffers and caches. */
239 
240         do {
241             bhrequest = 0;
242             uptodate = 1;
243             while (blocks) {
244                 --blocks;
245                 *bhb = getblk(dev, block++, blocksize);
246                 if (*bhb && !buffer_uptodate(*bhb)) {
247                     uptodate = 0;
248                     bhreq[bhrequest++] = *bhb;
249                 }
250 
251                 if (++bhb == &buflist[NBUF])
252                     bhb = buflist;
253 
254                 /* If the block we have on hand is uptodate, go ahead
255                    and complete processing. */
256                 if (uptodate)
```

```
257                     break;
258                 if (bhb == bhe)
259                     break;
260             }
261 
262             /* Now request them all */
263             if (bhrequest) {
264                 ll_rw_block(READ, bhrequest, bhreq);
265             }
266 
267             do { /* Finish off all I/O that has actually completed */
268                 if (*bhe) {
269                     wait_on_buffer(*bhe);
270                     if (!buffer_uptodate(*bhe)) {   /* read error? */
271                             brelse(*bhe);
272                         if (++bhe == &buflist[NBUF])
273                           bhe = buflist;
274                         left = 0;
275                         break;
276                     }
277                 }
278                 if (left < blocksize - offset)
279                     chars = left;
280                 else
281                     chars = blocksize - offset;
282                 *ppos += chars;
283                 left -= chars;
284                 read += chars;
285                 if (*bhe) {
286                     copy_to_user(buf,offset+(*bhe)->b_data,chars);
287                     brelse(*bhe);
288                     buf += chars;
289                 } else {
290                     while (chars-- > 0)
291                         put_user(0,buf++);
292                 }
293                 offset = 0;
294                 if (++bhe == &buflist[NBUF])
295                     bhe = buflist;
296             } while (left > 0 && bhe != bhb && (!*bhe || !buffer_locked(*bhe)));
297             if (bhe == bhb && !blocks)
298                 break;
299         } while (left > 0);
300 
301     /* Release the read-ahead blocks */
302         while (bhe != bhb) {
303             brelse(*bhe);
304             if (++bhe == &buflist[NBUF])
```

```
305                 bhe = buflist;
306         };
307         if (!read)
308             return -EIO;
309         filp->f_reada = 1;
310         return read;
311     }
```

读者对这段代码也许会感到似曾相识，实际上它的主体确实与读普通文件时调用的函数 block_read_full_page( )十分相似（见上册第 5 章“文件的写与读”一节）。开头部分关于记录块大小、关于预读的计算和处理等，读者在以前都看到过类似的代码。内核中有一些以块设备的主设备号为下标的指针数组，都是在 drivers/block/ll_rw_blk.c 中定义的。这些数组中的每个指针又指向另一个以次设备号为下标的数组，这样在逻辑上就等同于二维数组。但是每个以次设备号为下标的数组的大小却可以不同，所以比较节省空间。这里(186 行)用到的数组 blksize_size[ ][ ]中的内容为各个具体设备的记录块大小（见“文件系统的安装与拆卸”），而 blk_size[ ][ ]中的内容(196 行)则为各项具体设备中含有 1024 字节记录块的个数。所以，size 就是具体设备的总容量（以字节计）；而 left 则为该设备中剩下未读的字节数(206 行)，208 行又进一步将其调整成本次读操作剩下未读的字节数。212 行根据位移计算出起始块号，然后 213 行把整个设备内的位移调整为记录块内的位移，215 行又计算出本次读操作中要读的块数。最后，再基于预读的考虑和设备的总容量，再对本次操作中要读出的记录块数 blocks 作出调整。

接着就是针对本次读文件操作中要读出的每个记录块的 do_while 循环了。函数 getblk( )根据设备号与记录块号的杂凑值试图从相应的杂凑表队列中找到该记录块的缓冲区。如果找不到，就为之分配一个缓冲区并将其指针填入一个 buffer_head 结构指针数组 bhreq[ ]，准备从设备上读入这个记录块。我们已经在“文件的写与读”一节中读过这个函数的代码，这里不再重复，读者不妨回过去重温一下。这个函数根据目标设备的设备号和设备上的逻辑记录块号，先在记录块的杂凑表队列中寻找，如果找到就简单了，不用从设备上读入了。否则就要为之分配一个空闲的 buffer_head 数据结构连同大小相符的缓冲区(实际上是缓冲页面的一部分)。

在“文件的写和读”一节中我们讲过，文件的内容在内存中既按记录块缓冲，又按页面缓冲。文件的 inode 结构中维持着一个缓冲页面队列，当需要读入新的页面时就通过 create_page_buffers( )分配一个存储页面，再把这个页面划分成若干缓冲区，各自相当于一个记录块，同时分配相应数量的 buffer_head 数据结构，并使它们指向这些缓冲区。每个缓冲区都有个 buffer_head 数据结构，定义于 include/linux/fs.h：

```
209     /*
210      * Try to keep the most commonly used fields in single cache lines (16
211      * bytes) to improve performance.  This ordering should be
212      * particularly beneficial on 32-bit processors.
213      *
214      * We use the first 16 bytes for the data which is used in searches
215      * over the block hash lists (ie. getblk( ) and friends).
216      *
217      * The second 16 bytes we use for lru buffer scans, as used by
218      * sync_buffers( ) and refill_freelist( ).  -- sct
```

```
219     */
220     struct buffer_head {
221         /* First cache line: */
222         struct buffer_head *b_next; /* Hash queue list */
223         unsigned long b_blocknr;    /* block number */
224         unsigned short b_size;      /* block size */
225         unsigned short b_list;      /* List that this buffer appears */
226         kdev_t      b_dev;          /* device (B_FREE = free) */
227
228         atomic_t    b_count;        /* users using this block */
229         kdev_t      b_rdev;         /* Real device */
230         unsigned long b_state;      /* buffer state bitmap (see above) */
231         unsigned long b_flushtime;  /* Time when (dirty) buffer should be written */
232
233         struct buffer_head *b_next_free;/* lru/free list linkage */
234         struct buffer_head *b_prev_free;/* doubly linked list of buffers */
235         struct buffer_head *b_this_page;/* circular list of buffers in one page */
236         struct buffer_head *b_reqnext;  /* request queue */
237
238         struct buffer_head **b_pprev;   /* doubly linked list of hash-queue */
239         char * b_data;                  /* pointer to data block (512 byte) */
240         struct page *b_page;            /* the page this bh is mapped to */
241         void (*b_end_io)(struct buffer_head *bh, int uptodate); /* I/O completion */
242         void *b_private;                /* reserved for b_end_io */
243
244         unsigned long b_rsector;        /* Real buffer location on disk */
245         wait_queue_head_t b_wait;
246
247         struct inode *      b_inode;
248         struct list_head    b_inode_buffers;
                                        /* doubly linked list of inode dirty buffers */
249     };
```

结构中的指针 b_data 指向真正的“缓冲区”，即相应内存页面中的一部分。每个缓冲区通过其 buffer_head 数据结构链入到若干队列中：

(1) 通过指针 b_next 和双重指针 b_pprev 链入一个单链杂凑队列；

(2) 通过 b_next_free 和 b_prev_free 链入一个双链的 LRU 队列或空闲队列，字段 b_list 则说明目前缓冲区在哪一个 LRU 队列中；

(3) 通过指针 b_this_page 将属于同一页面的缓冲区链在一起，并通过指针 b_page 指向所属页面的 page 数据结构；

(4) 通过队列头 b_inode_buffers 链入所属文件的 inode 数据结构，并通过指针 b_inode 指向该 inode 数据结构。

此外，结构中的 b_dev 为目标设备的设备号；这个字段是在通过 getblk( )分配缓冲区时根据调用参数设置的，从 block_read( ) 的代码(183 和 245 行)可以看出这个设备号来自 inode 结构中的 i_rdev，那就是 inode 结构所代表文件所在的设备；当所代表的文件为块设备文件时，那就是目标设备的设备号。

有时候目标设备只是个逻辑设备，最后要落实到另一个设备上，此时用另一个字段 b_rdev 用于实际设备的设备号。

对普通文件而言，在文件层上的操作着眼于页面。相比之下，对块设备文件的访问则直接就跟缓冲区打交道，而跳过了页面这一层。所以直接就通过 getblk( )搜索或分配已经有了缓冲区的 buffer_head 数据结构。相比之下，getblk( )是个比较低层的函数，主要用于块设备上为文件系统的组织和管理所需的记录块，即所谓“meta data”，也可以说是供文件系统“内部使用”。例如，在对普通文件的读写中，如果文件比较大而需要间接映射，那就需要找到用于间接映射表的记录块，这就要调用 getblk( )了（见“文件的写和读”一节中 ext2_alloc_branch( )的代码）。又如，在 path_walk( )中要读入一个目录，即与一个目录节点相联系的记录块时，也要调用 getblk( )。这些记录块在逻辑上都不属于任何一个普通文件的内容，所以相应的存储页面不挂入任何 inode 结构中的缓冲页面队列。

回到 block_read( )的代码中，看一下调用 getblk( )时(245 行)使用的参数，就可以发现：作为设备上逻辑块号的 block 正是前面通过线性映射得到的文件内逻辑块号。也就是说二者是相同的。然而，对于普通文件，从文件内逻辑块号到设备上逻辑块号的映射却是相当复杂的。这正是块设备文件的文件层相比之下显得很单薄、很简单的根本原因。

从 getblk( )返回到 block_read( )以后，就可以知道是否真的需要从块设备上读入某个记录块了。但是，每次只从块设备上读入一个记录块是很不经济的。块设备通常是磁盘设备，它的介质在旋转，并且划分成“柱面”，在读出之前要先将磁头定位。由于块设备的这些特性，成片地读出逻辑上连续或邻近的记录块比分次读出个别的记录块有效得多。所以，block_read( )通过数组 bhreq[ ]积累起成片读入请求，具体的办法是：

(1) 如果下一个记录块，即本次 do-while 循环中的第一个记录块，不需要从设备上读入，那么这个 while 循环马上就结束了（257 行），此时 bhrequest 为 0，所以跳过 ll_rw_block( )不从设备上读入。数组 buflist[ ]用来收集和积累所需的记录块缓冲区(见 216 行和 245 行)，其中有些不需要从设备上读入(如果 246 行的条件能满足)，有些则需要从设备上读入。

(2) 反之，如果本次 do-while 循环中的第一个记录块需要从设备上读入，那就要通过 bhreq[ ]积累起成片的读入请求，但不超过数组 buflist[ ]的大小。第 243 行至 260 行的 while 循环的作用，就是在数组 buflist[ ]中积累起需要从块设备读入的记录块缓冲区，同时在 bhreq[ ]中积累起成片的读入请求，直至把 buflist[ ]填满或者 blocks 变成了 0 为止。对数组 buflist[ ]是按循环缓冲区的方式使用的(见 251～252 行以及 258～259 行)，每次进入 243 行的 while 循环时 bhb 和 bhe 相同，开始循环以后，当二者再次相同时，就说明 buflist[ ]中已经填满了(258 行)。积累了成片的记录块缓冲区以后，就通过 ll_rw_block( )启动块设备的成片读入。数组 bhreq[ ]的大小与 buflist[ ]相同，在最坏的情况下所有的缓冲区都需要从设备上读入。但也可能有几个缓冲区不需要读入，从而不出现在 bhreq[ ]中。

(3) 然后，通过一个内层 do_while 循环(267 行)处理 buflist[ ]中积累起的下一片缓冲区，处理完以后又回到外层 do-while 循环的开头(240 行)，直至全部完成。

至此，对块设备文件和对普通文件的读操作终于殊途同归，都归结成了对 ll_rw_block( )的调用，这才真正进入了设备驱动层。顾名思义，ll_rw_block( )的作用是“低层块设备读写”。

类似地，对块设备的写操作 block_write( )最终也是对 ll_rw_block( )的调用。不过块设备的写操作常常需要先从设备上读入一个记录块，加以修改以后再把它写回去。我们把 block_write( )的代码留给

读者。

在“文件的写和读”一节中，读者已经看到对普通文件的写操作并不直接启动块设备的写操作，而是在改变了记录块缓冲区的内容后就将其提交给一个内核线程 kflushd（其执行程序为 bdflush( )）。这个线程平时都在睡眠。一旦需要“冲刷”到块设备上的页面积累到一定的数量（以及在其他一些条件下）时就将它唤醒，使它执行一个函数 flush_dirty_buffers( )。可想而知，flush_dirty_buffers( )也调用 ll_rw_block( )启动对具体块设备的写操作。函数 flush_dirty_buffers( )的代码在文件 buffer.c 中：

[bdflush( ) > flush_dirty_buffers( )]

```
2537    /* This is the _only_ function that deals with flushing async writes
2538       to disk.
2539       NOTENOTENOTENOTE: we _only_ need to browse the DIRTY lru list
2540       as all dirty buffers lives _only_ in the DIRTY lru list.
2541       As we never browse the LOCKED and CLEAN lru lists they are infact
2542       completly useless. */
2543    static int flush_dirty_buffers(int check_flushtime)
2544    {
2545        struct buffer_head * bh, *next;
2546        int flushed = 0, i;
2547
2548     restart:
2549        spin_lock(&lru_list_lock);
2550        bh = lru_list[BUF_DIRTY];
2551        if (!bh)
2552            goto out_unlock;
2553        for (i = nr_buffers_type[BUF_DIRTY]; i-- > 0; bh = next) {
2554            next = bh->b_next_free;
2555
2556            if (!buffer_dirty(bh)) {
2557                __refile_buffer(bh);
2558                continue;
2559            }
2560            if (buffer_locked(bh))
2561                continue;
2562
2563            if (check_flushtime) {
2564                /* The dirty lru list is chronologically ordered so
2565                   if the current bh is not yet timed out,
2566                   then also all the following bhs
2567                   will be too young. */
2568                if (time_before(jiffies, bh->b_flushtime))
2569                    goto out_unlock;
2570            } else {
2571                if (++flushed > bdf_prm.b_un.ndirty)
2572                    goto out_unlock;
2573            }
```

```
2574
2575            /* OK, now we are committed to write it out. */
2576            atomic_inc(&bh->b_count);
2577            spin_unlock(&lru_list_lock);
2578            ll_rw_block(WRITE, 1, &bh);
2579            atomic_dec(&bh->b_count);
2580
2581            if (current->need_resched)
2582                schedule( );
2583            goto restart;
2584        }
2585    out_unlock:
2586        spin_unlock(&lru_list_lock);
2587
2588        return flushed;
2589    }
```

系统中有个缓冲区队列的数组 lru_list[ ]，这个数组以缓冲区的类型为下标，所以 lru_list[BUF_DIRTY]就是其中已经“脏”了的缓冲区的队列。除此以外，数组中还有 BUF_CLEAN、BUF_LOCKED、BUF_PROTECTED 等队列。所有的缓冲区都按 LRU，即最后一次受到访问的时间先后排在其中的某个队列中。另一个数组 nr_buffers_type[ ]与之对应，记录着每个队列的大小。这个函数扫描脏缓冲区队列，依次考察队列中的页面。如果一个页面虽然在 BUF_DIRTY 队列中，但是实际上已经不再是“脏”的了，那就通过__refile_buffer( )把它转移到其他队列中。如果一个缓冲区已经加了锁就把它跳过。否则，就调用 ll_rw_block( )把它写入设备，写入设备以后这个缓冲区就变成干净的了。不过，这种操作不一定是对队列中所有缓冲区的，如果参数 check_flushtime 非 0，则表示仅对已经超时的缓冲区进行处理。由于队列中所有的缓冲区都是按“年龄”排列的，所以在碰到第一个尚未超时的缓冲区时循环就结束了。还有，由于这整个过程可能需要较长的时间，也不一定是在系统调用中执行，所以在每次调用 ll_rw_block( )以后都要检查是否需要调度(通常都是从系统空间返回到用户空间的前夕才检查)，如果需要就主动调用 schedule( )加以调度。以后当再次被调度运行时则回到 restart 处(2548 行)重新扫描整个队列，因为情况可能已经改变。

此外，系统中还有一个内核线程 kupdate( )，它管的事更宽，包括超级块的同步、索引节点的同步，也包括缓冲区的同步（即冲刷），还包括对 tq_disk 任务队列（见第 3 章）的执行。它也通过 flush_dirty_buffers( )冲刷内容已经改变的缓冲区，所以最终也是调用 ll_rw_block( )。

由此可见，无论是对普通文件的读/写，还是对块设备文件的读/写，最终都是通过 ll_rw_block( )完成的。与这个函数并立的还有 ll_rw_block_locked( )，这两个函数的代码在 drivers/block/ll_rw_blk.c 中。这里要再次强调，对 ll_rw_block( )的调用路径是比较多的，我们随同代码列出的只是其中之一。

[sys_read( ) > block_read( ) > ll_rw_block( )]

```
 987    /**
 988     * ll_rw_block: low-level access to block devices
 989     * @rw: whether to %READ or %WRITE or maybe %READA (readahead)
 990     * @nr: number of &struct buffer_heads in the array
```

```
991      * @bhs: array of pointers to &struct buffer head
992      *
993      * ll_rw_block( ) takes an array of pointers to &struct buffer_heads,
994      * and requests an I/O operation on them, either a %READ or a %WRITE.
995      * The third %READA option is described in the documentation for
996      * generic_make_request( ) which ll_rw_block( ) calls.
997      *
998      * This function provides extra functionality that is not in
999      * generic_make_request( ) that is relevant to buffers in the buffer
1000     * cache or page cache.  In particular it drops any buffer that it
1001     * cannot get a lock on (with the BH_Lock state bit), any buffer that
1002     * appears to be clean when doing a write request, and any buffer that
1003     * appears to be up-to-date when doing read request.  Further it marks
1004     * as clean buffers that are processed for writing (the buffer cache
1005     * wont assume that they are actually clean until the buffer gets
1006     * unlocked).
1007     *
1008     * ll_rw_block sets b_end_io to simple completion handler that marks
1009     * the buffer up-to-date (if approriate), unlocks the buffer and wakes
1010     * any waiters.  As client that needs a more interesting completion
1011     * routine should call submit_bh( ) (or generic_make_request( ))
1012     * directly.
1013     *
1014     * Caveat:
1015     *  All of the buffers must be for the same device, and must also be
1016     *  of the current approved size for the device.  */
1017
1018     void ll_rw_block(int rw, int nr, struct buffer_head * bhs[ ])
1019     {
1020         unsigned int major;
1021         int correct_size;
1022         int i;
1023
1024         major = MAJOR(bhs[0]->b_dev);
1025
1026         /* Determine correct block size for this device. */
1027         correct_size = BLOCK_SIZE;
1028         if (blksize_size[major]) {
1029             i = blksize_size[major][MINOR(bhs[0]->b_dev)];
1030             if (i)
1031                 correct_size = i;
1032         }
1033
1034         /* Verify requested block sizes. */
1035         for (i = 0; i < nr; i++) {
1036             struct buffer_head *bh;
1037             bh = bhs[i];
1038             if (bh->b_size != correct_size) {
```

```
1039                printk(KERN_NOTICE "ll_rw_block: device %s: "
1040                       "only %d-char blocks implemented (%u)\n",
1041                       kdevname(bhs[0]->b_dev),
1042                       correct_size, bh->b_size);
1043                goto sorry;
1044            }
1045        }
1046
1047        if ((rw & WRITE) && is_read_only(bhs[0]->b_dev)) {
1048            printk(KERN_NOTICE "Can't write to read-only device %s\n",
1049                   kdevname(bhs[0]->b_dev));
1050            goto sorry;
1051        }
1052
1053        for (i = 0; i < nr; i++) {
1054            struct buffer_head *bh;
1055            bh = bhs[i];
1056
1057            /* Only one thread can actually submit the I/O. */
1058            if (test_and_set_bit(BH_Lock, &bh->b_state))
1059                continue;
1060
1061            /* We have the buffer lock */
1062            bh->b_end_io = end_buffer_io_sync;
1063
1064            switch(rw) {
1065            case WRITE:
1066                if (!atomic_set_buffer_clean(bh))
1067                    /* Hmmph! Nothing to write */
1068                    goto end_io;
1069                __mark_buffer_clean(bh);
1070                break;
1071
1072            case READA:
1073            case READ:
1074                if (buffer_uptodate(bh))
1075                    /* Hmmph! Already have it */
1076                    goto end_io;
1077                break;
1078            default:
1079                BUG( );
1080        end_io:
1081                bh->b_end_io(bh, test_bit(BH_Uptodate, &bh->b_state));
1082                continue;
1083            }
1084
1085            submit_bh(rw, bh);
1086        }
```

```
1087        return;
1088
1089    sorry:
1090        /* Make sure we don't get infinite dirty retries.. */
1091        for (i = 0; i < nr; i++)
1092            mark_buffer_clean(bhs[i]);
1093    }
```

参数 bhs[ ]为一个缓冲区头部 buffer_head 的指针数组，数组中的每个指针都指向一个需要从设备读/写的记录块的 buffer_head 数据结构；参数 nr 则为该数组的大小。数组中指定要读/写的记录块不必是连续的(不过通常是邻近的)，但是必须在同一设备上。参数 rw 指明了要进行的操作。

每种块设备就好像是个服务器，都有一个操作请求队列，队列的头部是一个 request_queue_t 数据结构，有关的定义见 include/linux/blkdev.h：

```
74    struct request_queue
75    {
76        /*
77         * the queue request freelist, one for reads and one for writes
78         */
79        struct list_head    request_freelist[2];
80
81        /*
82         * Together with queue_head for cacheline sharing
83         */
84        struct list_head    queue_head;
85        elevator_t        elevator;
86
87        request_fn_proc      * request_fn;
88        merge_request_fn     * back_merge_fn;
89        merge_request_fn     * front_merge_fn;
90        merge_requests_fn    * merge_requests_fn;
91        make_request_fn      * make_request_fn;
92        plug_device_fn       * plug_device_fn;
93        /*
94         * The queue owner gets to use this for whatever they like.
95         * ll_rw_blk doesn't touch it.
96         */
97        void             * queuedata;
98
99        /*
100        * This is used to remove the plug when tq_disk runs.
101        */
102       struct tq_struct    plug_tq;
103
104       /*
105        * Boolean that indicates whether this queue is plugged or not.
```

```
106         */
107         char            plugged;
108
109         /*
110          * Boolean that indicates whether current_request is active or
111          * not.
112          */
113         char            head_active;
114
115         /*
116          * Is meant to protect the queue in the future instead of
117          * io request_lock
118          */
119         spinlock_t      request_lock;
120
121         /*
122          * Tasks wait here for free request
123          */
124         wait_queue_head_t   wait_for_request;
125     };
```

```
11      typedef struct request_queue request_queue_t;
```

数据结构中各个字段的作用和意义随着代码的进展会变得清楚起来。结构中从 request_fn 到 plug_device_fn 都是一些函数指针，例如第 87 行的意思是说：request_fn 是一个指针，指向类型为 request_fn_proc 的对象。而 request_fn_proc 则通过#typedef 定义为一种函数（见 blkdev.h）：

```
63      typedef void (request_fn_proc) (request_queue_t *q);
```

其余的函数指针也与此类似，这些指针（连同其他字段）都是在相应块设备初始化时设置好的。需要对一个块设备进行操作时，就为之设置好一个数据结构（见后）并将其挂入相应的请求队列。

为了把各种块设备的操作请求队列有效地组织起来，内核中还设置了一个结构数组 blk_dev[ ]，见 driver/block/ll_rw_blk.c：

```
72      /* blk_dev_struct is:
73       *  *request_fn
74       *  *current_request
75       */
76      struct blk_dev_struct blk_dev[MAX_BLKDEV];
                                                /* initialized by blk_dev_init( ) */
```

这个数组以主设备号为下标，数组中的每个元素都是一个 blk_dev_struct 数据结构，其定义在 blkdev.h 中给出：

```
127     struct blk_dev_struct {
```

```
128         /*
129          * queue_proc has to be atomic
130          */
131         request_queue_t      request_queue;
132         queue_proc       *queue;
133         void             *data;
134     };
```

它的主体就是（操作）请求队列 request_queue。此外，结构中还有一个函数指针 queue，当这个指针为非 0 时就调用这个函数来找到具体设备的请求队列，这是考虑具有同一主设备号的多项设备而设的。这个指针也在设备初始化时设置好，通常当该指针非 0 时还要使用数据结构中的另一个指针 data 来提供辅助性的信息，以帮助该函数找到特定设备的请求队列。

进入 ll_rw_block( )以后，先对记录块大小作一些检查；然后，如果是写访问，则还要检查目标设备是否可写。内核中有个二维数组 ro_bits，定义于 drivers/block/ll_rw_blk.c：

```
538     static long ro_bits[MAX_BLKDEV][8];
```

每个设备在这数组中都有个标志位，通过系统调用 ioctl( )可以将一个标志位设置成 1 或 0，表示相应设备为只读或可写，而 is_read_only( )则根据设备号检查这个数组中的标志位是否为 1。

接下去，就是通过 1053 行的 for 循环依次处理对各个缓冲区的读写请求了。对于要读写的每个记录块，首先将其缓冲区加上锁，还要将其 buffer_head 结构中的函数指针 b_end_io 设置成指向 end_buffer_io_sync( )。当完成对给定记录块的读写时，就调用这个函数。此外，对于待写的缓冲区，其 BH_Dirty 标志位应该是 1，否则就不需要写了，而既然写了就要把它请成 0，并通过 __mark_buffer_clean( )将缓冲区转移到干净页面的LRU 队列中。反之，对于待读的缓冲区，则其 Uptodate 标志位应该是 0，否则就不需要读了。每个具体的设备就好像是个服务器，所以最后具体的读写是通过 submit_bh( )将读写请求提交给"服务器"完成的，每次一个记录块，其代码在 drivers/block/ll_rw_blk.c 中：

[sys_read( ) > block_read( ) > ll_rw_block( ) > submit_bh( )]

```
939     /**
940      * submit_bh: submit a buffer_head to the block device later for I/O
941      * @rw: whether to %READ or %WRITE, or mayve to %READA (read ahead)
942      * @bh: The &struct buffer_head which describes the I/O
943      *
944      * submit_bh( ) is very similar in purpose to generic make request( ), and
945      * uses that function to do most of the work.
946      *
947      * The extra functionality provided by submit_bh is to determine
948      * b_rsector from b_blocknr and b_size, and to set b_rdev from b_dev.
949      * This is is appropriate for IO requests that come from the buffer
950      * cache and page cache which (currently) always use aligned blocks.
951      */
952     void submit_bh(int rw, struct buffer head * bh)
```

```
953     {
954         if (!test_bit(BH_Lock, &bh->b_state))
955             BUG( );
956
957         set_bit(BH_Req, &bh->b_state);
958
959         /*
960          * First step, 'identity mapping' - RAID or LVM might
961          * further remap this.
962          */
963         bh->b_rdev = bh->b_dev;
964         bh->b_rsector = bh->b_blocknr * (bh->b_size>>9);
965
966         generic_make_request(rw, bh);
967
968         switch (rw) {
969             case WRITE:
970                 kstat.pgpgout++;
971                 break;
972             default:
973                 kstat.pgpgin++;
974                 break;
975         }
976     }
```

对具体记录块缓冲区的操作请求必须是独占的，所以在调用 submit_bh( )之前必须已经对缓冲区通过其 BH_LOCK 标志位加了锁(见前面的 1058 行)，这里则通过检验该标志位来加以验证。进一步，由一个进程提交的操作请求未完成之前，是不允许其他进程再来提交对同一记录块缓冲区的操作请求的。所以，在提交对一特定记录块缓冲区的操作请求之前，还要通过标志位 BH_Req 标志位加锁。

显然，操作的主体是 generic_make_request( )。对于一般的设备，实际的目标设备 bh->b_rdev 就是逻辑的目标设备 bh->b_dev，而设备上的扇区号 bh->b_rsector 则要根据目标块号 bh->b_blocknr 换算，因为扇区的大小通常为 512 字节，而记录块的大小为 1024 字节。值得注意的是，在文件系统层次上这里的记录块号已经是所谓“物理块号”了，但是到了设备驱动层这仍是逻辑意义上的操作对象，而物理意义上的对象是扇区。

函数generic_make_request( )的代码在drivers/block/ll_rw_blk.c中：

[sys_read( ) > block_read( ) > ll_rw_block( ) > submit_bh( ) > generic_make_request( )]

```
850     /**
851      * generic_make_request: hand a buffer head to it's device driver for I/O
852      * @rw:  READ, WRITE, or READA - what sort of I/O is desired.
853      * @bh:  The buffer head describing the location in memory and on the device.
854      *
855      * generic_make_request( ) is used to make I/O requests of block
856      * devices. It is passed a &struct buffer_head and a &rw value.  The
```

```
857      * %READ and %WRITE options are (hopefully) obvious in meaning.  The
858      * %READA value means that a read is required, but that the driver is
859      * free to fail the request if, for example, it cannot get needed
860      * resources immediately.
861      *
862      * generic_make_request( ) does not return any status.  The
863      * success/failure status of the request, along with notification of
864      * completion, is delivered asynchronously through the bh->b_end_io
865      * function described (one day) else where.
866      *
867      * The caller of generic make request must make sure that b_page,
868      * b addr, b size are set to describe the memory buffer, that b_rdev
869      * and b_rsector are set to describe the device address, and the
870      * b_end_io and optionally b_private are set to describe how
871      * completion notification should be signaled.  BH_Mapped should also
872      * be set (to confirm that b_dev and b_blocknr are valid).
873      *
874      * generic make request and the drivers it calls may use b_reqnext,
875      * and may change b_rdev and b_rsector.  So the values of these fields
876      * should NOT be depended on after the call to generic_make_request.
877      * Because of this, the caller should record the device address
878      * information in b_dev and b_blocknr.
879      *
880      * Apart from those fields mentioned above, no other fields, and in
881      * particular, no other flags, are changed by generic make request or
882      * any lower level drivers.
883      * */
884     void generic_make_request (int rw, struct buffer_head * bh)
885     {
886         int major = MAJOR(bh->b_rdev);
887         request_queue_t *q;
888
889         if (!bh->b_end_io) BUG( );
890         if (blk_size[major]) {
891             unsigned long maxsector = (blk size[major][MINOR(bh->b rdev)] << 1) + 1;
892             unsigned int sector, count;
893
894             count = bh->b_size >> 9;
895             sector = bh->b_rsector;
896
897             if (maxsector < count || maxsector - count < sector) {
898                 bh->b_state &= (1 << BH_Lock) | (1 << BH_Mapped);
899                 if (blk_size[major][MINOR(bh->b_rdev)]) {
900
901                     /* This may well happen - the kernel calls bread( )
902                        without checking the size of the device, e.g.,
903                        when mounting a device. */
904                     printk(KERN_INFO
```

```
905                             "attempt to access beyond end of device\n");
906                     printk(KERN_INFO "%s: rw=%d, want=%d, limit=%d\n",
907                             kdevname(bh->b_rdev), rw,
908                             (sector + count)>>1,
909                             blk_size[major][MINOR(bh->b_rdev)]);
910                 }
911                 bh->b_end_io(bh, 0);
912                 return;
913             }
914         }
915 
916         /*
917          * Resolve the mapping until finished. (drivers are
918          * still free to implement/resolve their own stacking
919          * by explicitly returning 0)
920          */
921         /* NOTE: we don't repeat the blk_size check for each new device.
922          * Stacking drivers are expected to know what they are doing.
923          */
924         do {
925             q = blk_get_queue(bh->b_rdev);
926             if (!q) {
927                 printk(KERN_ERR
928 "generic_make_request: Trying to access nonexistent block-device %s(%ld)\n",
929                         kdevname(bh->b_rdev), bh->b_rsector);
930                 buffer_IO_error(bh);
931                 break;
932             }
933 
934         }
935         while (q->make_request_fn(q, rw, bh));
936     }
```

如前所述，数组 blk_size[ ][ ]中的内容为各项具体设备中含有 1024 字节记录块的个数。这个数组是在 drivers/block/ll_rw_blk.c 中定义的：

```
78    /*
79     * blk_size contains the size of all block-devices in units of 1024 byte
80     * sectors:
81     *
82     * blk_size[MAJOR][MINOR]
83     *
84     * if (!blk_size[MAJOR]) then no minor size checking is done.
85     */
86    int * blk_size[MAX_BLKDEV];
```

注意 blk_size[ ]实际上是个 int 指针数组，以主设备号为下标，如果某个主设备号所对应的设备存

在就指向一个以次设备号为下标的 int 数组，否则就为 0。用这样的方式实现“稀疏”数组，即有很多元素为 0 的数组，可以大大节省空间，而运行效率的下降却并不显著，所以在 Linux 内核中常常要用到。代码中的 890-914 行检查记录块的扇区范围，记录块的起点扇区号为 bh->b_rsector，扇区的个数等于记录块的大小 bh->b_size 除于 512。如果越出了范围就通过 printk( )在系统的运行日志中登记出错信息，然后返回。

接着要根据设备号找到目标设备的操作请求队列，这是由 blk_get_queue( )完成的，其代码在 drivers/block/ll_rw_blk.c 中

[sys_read( ) > block_read( ) > ll_rw_block( ) > submit_bh( ) > generic_make_request( ) > blk_get_queue( )]

```
138     /*
139      * NOTE: the device-specific queue( ) functions
140      * have to be atomic!
141      */
142     request_queue_t *blk_get_queue(kdev_t dev)
143     {
144         request_queue_t *ret;
145         unsigned long flags;
146 
147         spin_lock_irqsave(&io_request_lock, flags);
148         ret = __blk_get_queue(dev);
149         spin_unlock_irqrestore(&io_request_lock, flags);
150 
151         return ret;
152     }
```

[sys_read( ) > block_read( ) > ll_rw_block( ) > submit_bh( ) > generic_make_request( ) > blk_get_queue( ) > __blk_get_queue( )]

```
128     static inline request_queue_t *__blk_get_queue(kdev_t dev)
129     {
130         struct blk_dev_struct *bdev = blk_dev + MAJOR(dev);
131 
132         if (bdev->queue)
133             return bdev->queue(dev);
134         else
135             return &blk_dev[MAJOR(dev)].request_queue;
136     }
```

以主设备号为下标，就可以在 blk_dev[ ]中找到目标设备的 blk_dev_struct 结构。对一种设备第一次调用__blk_get_queue( )时，要通过其函数指针 queue 找到这种设备的读写请求队列，找到了以后就把它记录在结构中的 request_queue 字段，以后就简单了。对于 IDE 硬盘，其 blk_dev_struct 结构中的函数指针 queue 在初始化时设置成指向 ide_get_queue( )，而且其指针 data 设置成指向代表着相应硬件接口的 ide_hwif_t 数据结构。所以，对于 IDE 硬盘，将会调用 ide_get_queue( )取得其操作请求队列，

这个函数的代码在 drivers/ide/ide.c 中：

[sys_read( ) > block_read( ) > ll_rw_block( ) > submit_bh( ) > generic_make_request( )
 > blk_get_queue( ) > __blk_get_queue( ) > ide_get_queue( )]

```
1367    /*
1368     * ide get queue( ) returns the queue which corresponds to a given device.
1369     */
1370    request_queue_t *ide_get_queue (kdev_t dev)
1371    {
1372        ide_hwif_t *hwif = (ide_hwif_t *)blk_dev[MAJOR(dev)].data;
1373
1374        return &hwif->drives[DEVICE NR(dev) & 1].queue;
1375    }
```

这里的宏操作 DEVICE_NR(dev)从设备号 dev 中取出次设备号，然后再取其最低位，这保证了次设备号只能取 0 和 1 两种值。每种 IDE 设备，更确切地说是每个 IDE 接口，都有个 ide_hwif_t 数据结构，后面我们会看到它的定义，但是这里从代码中已可看到这个结构中有个 drives[ ]数组，对应着可以连接到同一 IDE 接口上的同种设备，每个这样的设备都有个读写请求队列。这是一个 request_queue_t 数据结构，前面我们已经列出了它的定义。

找到了具体的队列以后，就要通过由这个队列提供的 make_request_fn 操作创建一个读写请求数据结构，并把它挂入该队列。在某些情况下，有些逻辑上存在的块设备可能并没有直接的对应物，而只是间接地映射到其他块设备上。这种仅在逻辑上存在的块设备只是一种中间的过渡层，因此要提供一个函数来实现相应的映射或完成对最终设备的操作，并在初始化阶段将其操作请求队列结构中的函数指针 make_reguest_fn 指向这个函数。如果这个函数所完成的只是从一种设备到另一种设备的映射，就返回一个正整数，使 924～935 行的 do-while 循环再执行一遍。而如果完成的是对终极设备的操作，则返回 0，从而结束该 do-while 循环。举例来说，在有容错要求的系统中可以在逻辑上设立一个“可靠硬盘”，并且将文件系统建立在可靠硬盘上，而实际上则把数据平行地存储在两个硬盘上。此时就可以编写一个专用于这种可靠硬盘的驱动程序，并使该设备的操作队列结构中的函数指针 make_request_fn 指向这个函数。当提交一个读写请求时，首先使 buffer_head 结构中的 b_rdev 字段指向第一个硬盘，找到第一个硬盘的读写请求队列(见 925 行，注意这里用的是 b_rdev 中的设备号)，并在“可靠硬盘”的 make_request_fn 中为第一个硬盘创建一个读写请求，然后将 buffer_head 结构中的 b_rdev 字段改成指向第二个硬盘，并返回 1。这样，就会再执行一遍 924～935 行的 do-while 循环，为第二个硬盘也创建一个读写请求，但是这一次 make_request_fn 操作返回 0，从而结束这个 do-while 循环。于是，对虚拟的“可靠硬盘”的读写请求就转化成了对两个具体硬盘的读写请求。至于我们在本节所关心的 IDE 硬盘，它本来就已经是终极设备，所以这个函数返回 0。对于普通的 IDE 硬盘，这个函数__make_request( )是完成提交操作请求的主体，适用于常规的（终极）块设备。它的主要任务就是为数据缓冲区准备一个操作请求，即 request 数据结构，并将这个数据结构挂入设备的操作请求队列中。如果在此之前该队列是空的，就要启动设备的输入/输出操作。一旦启动以后，对整个队列的操作就是由中断驱动的了。这个由中断驱动的过程一直要到队列中再没有等待着操作的请求时才结束。如果在将操作请求挂入队列之前队列中非空，那就说明由中断驱动的整个过程已经启动，从概念上说只要把新的请求挂在队列

的尾部就可以了。但是，实际上常常要进行某种程度的优化。这个函数所作的优化之一，是在队列中由后向前搜索，试图将新的请求与已经在队列中的其他请求合并。合并的条件主要是操作相同（同为读或写）并且扇区连续。有时候，两个请求的扇区相连续，但是缓冲区却不连续（分属不同的页面），此时两个请求仍能合并，但是形成两个不连续的缓冲区“分段”（segment），这是允许的。不过，对于每个 request 结构中可以容纳的分段数量有个限制。在 request 结构中有个 buffer_head 结构的队列，所谓合并就是将多个 buffer_head 结构连在一起成为一个操作请求。由于这个函数的代码较长，我们分段阅读，其代码在 drivers/block/ll_rw_blk.c 中：

[sys_read( ) > block_read( ) > ll_rw_block( ) > submit_bh( ) > generic_make_request( )
 > __make_request( )]

```
695	static int __make_request(request_queue_t * q, int rw,
696				  struct buffer_head * bh)
697	{
698		unsigned int sector, count;
699		int max_segments = MAX_SEGMENTS;
700		struct request * req = NULL, *freereq = NULL;
701		int rw_ahead, max_sectors, el_ret;
702		struct list_head *head;
703		int latency;
704		elevator_t *elevator = &q->elevator;
705	
706		count = bh->b_size >> 9;
707		sector = bh->b_rsector;
708	
709		rw_ahead = 0;	/* normal case; gets changed below for READA */
710		switch (rw) {
711			case READA:
712				rw_ahead = 1;
713				rw = READ;	/* drop into READ */
714			case READ:
715			case WRITE:
716				break;
717			default:
718				BUG( );
719				goto end_io;
720		}
721	
722		/* We'd better have a real physical mapping!
723		   Check this bit only if the buffer was dirty and just locked
724		   down by us so at this point flushpage will block and
725		   won't clear the mapped bit under us. */
726		if (!buffer_mapped(bh))
727			BUG( );
728	
729		/*
```

```
730          * Temporary solution - in 2.5 this will be done by the lowlevel
731          * driver. Create a bounce buffer if the buffer data points into
732          * high memory - keep the original buffer otherwise.
733          */
734     #if CONFIG_HIGHMEM
735         bh = create_bounce(rw, bh);
736     #endif
737
```

所请求的操作种类可以是 READ、WRITE 以及 READA 三者之一。如果是 READA，即预读，则将预读标志 rw_ahead 设置成 1，并将操作种类改成 READ。记录块的缓冲区必须属于某个缓冲页面，其 buffer_head 结构中的 Mapped 标志位应该为 1，所以这里通过 buffer_mapped( )加以确认。如果 CPU 支持超过 32 位（4GB）内存空间（HIGNMEM），而缓冲区又在高于 4GB 的空间中，就要通过 create_bounce( )为其在 4GB 以内建立一个镜像，而具体的读写将针对这个镜像进行，完成以后再把其内容复制到高于 4GB 的空间中。这是因为目前 DMA 控制器的寻址能力都不超过 32 位。

继续往下看。

[sys_read( ) > block_read( ) > ll_rw_block( ) > submit_bh( ) > generic_make_request( ) > __make_request( )]

```
738     /* look for a free request. */
739         /*
740          * Try to coalesce the new request with old requests
741          */
742         max_sectors = get_max_sectors(bh->b_rdev);
743
744         latency = elevator_request_latency(elevator, rw);
745
746         /*
747          * Now we acquire the request spinlock, we have to be mega careful
748          * not to schedule or do something nonatomic
749          */
750     again:
751         spin_lock_irq(&io_request_lock);
752
753         /*
754          * skip first entry, for devices with active queue head
755          */
756         head = &q->queue_head;
757         if (q->head_active && !q->plugged)
758             head = head->next;
759
760         if (list_empty(head)) {
761             q->plug_device_fn(q, bh->b_rdev); /* is atomic */
762             goto get_rq;
763         }
```

```
764
```

这里调用了一个函数 elevator_request_latency( )计算出一个数值 latency，那是为后面的另一种优化进行一些准备，我们暂时放一下。我们也暂时跳过这里的 757～758 行。

如果队列原来是空的，那就有个启动 I/O 操作的问题。当然，I/O 的启动要到将第一个请求挂入队列中以后才进行。可是由谁进行呢？一个选择是由当前进程自己直接调用一个函数来启动。另一个选择是将 I/O 的启动作为一个 bottom_half 函数（见第 3 章）“插入”内核中的一个任务队列 tq_disk。这样，每当执行这个任务队列时，就会依次执行队列中等待执行的所有 bottom_half 函数，包括具体块设备对 I/O 的启动。这里（761 行）通过函数指针 plug_device_fn 调用一个由具体队列提供的函数，这个函数可以决定是否将队列插入 tq_disk。就 IDE 硬盘而言，这个指针在初始化时设置成指向 generic_plug_device( )，它将设备的操作请求队列插入 tq_disk 中。代表着操作请求队列的 request_queue_t 数据结构中有一个成分 plug_tq，是一个 tq_struct 结构，就是为链入 tq_disk 队列而设的。还有一个成分 plugged，是一个标志，当这个标志为 1 时就表示该数据结构已经在 tq_disk 队列中。当执行任务队列 tq_disk 时，会通过队列中每个 tq_struct 结构里的函数指针 routine 执行一个 bottom-half 函数，启动具体硬盘的 I/O。将操作请求队列插入 tq_disk 以后，就直接转到 get_rq 标号处，进一步处理将操作请求加入请求队列的事。但是，如果操作请求队列原来不是空的话，那就要先试试是否能将新的请求与队列中原有的请求合在一起进行一些优化。

继续往下看__make_request( )的代码，这一段就是对优化的尝试。

[sys_read( ) > block_read( ) > ll_rw_block( ) > submit_bh( ) > generic_make_request( ) > __make_request( )]

```
765         el_ret = elevator->elevator_merge_fn(q, &req, bh, rw,
766                                 &max_sectors, &max_segments);
767         switch (el_ret) {
768 
769             case ELEVATOR_BACK_MERGE:
770                 if (!q->back_merge_fn(q, req, bh, max_segments))
771                     break;
772                 req->bhtail->b_reqnext = bh;
773                 req->bhtail = bh;
774                 req->nr_sectors = req->hard_nr_sectors += count;
775                 req->e = elevator;
776                 drive_stat_acct(req->rq_dev, req->cmd, count, 0);
777                 attempt_back_merge(q, req, max_sectors, max_segments);
778                 goto out;
779 
780             case ELEVATOR_FRONT_MERGE:
781                 if (!q->front_merge_fn(q, req, bh, max_segments))
782                     break;
783                 bh->b_reqnext = req->bh;
784                 req->bh = bh;
785                 req->buffer = bh->b_data;
786                 req->current_nr_sectors = count;
787                 req->sector = req->hard_sector = sector;
```

```
788                 req->nr_sectors = req->hard_nr_sectors += count;
789                 req->e = elevator;
790                 drive_stat_acct(req->rq_dev, req->cmd, count, 0);
791                 attempt_front_merge(q, head, req, max_sectors, max_segments);
792                 goto out;
793             /*
794              * elevator says don't/can't merge. get new request
795              */
796             case ELEVATOR_NO_MERGE:
797                 break;
798 
799             default:
800                 printk("elevator returned crap (%d)\n", el_ret);
801                 BUG( );
802         }
803 
```

当一个 IDE 设备的队列中已经有操作请求在等待，而又有新的操作请求到来时，可以作两种不同的优化。一种是因扇区连续而引起的操作合并，另一种是对操作路线所作的优化或者说调度。为此，在 request_queue_t 结构内部设立了一个 elevator_t 数据结构，名为 elevator，此命名从字义上就表明了对磁盘操作的调度类似于对电梯的调度。这种数据结构的定义在 include/linux/elevator.h 和 include/linux/blkdev.h 中：

```
15    struct elevator_s
16    {
17        int sequence;
18 
19        int read_latency;
20        int write_latency;
21        int max_bomb_segments;
22 
23        unsigned int nr_segments;
24        int read_pendings;
25 
26        elevator_fn * elevator_fn;
27        elevator_merge_fn *elevator_merge_fn;
28        elevator_dequeue_fn *dequeue_fn;
29 
30        unsigned int queue_ID;
31    };
```

```
13    typedef struct elevator_s elevator_t;
```

这个数据结构中提供了几个函数指针用于操作的优化，所以特定的 elevator_t 决定了具体的优化算法。目前的 Linux 内核只提供一种(也许应该说一套) 优化算法，那就是 ELEVATOR_LINUS，定义于

include/linux/elevator.h：

```
 99    #define ELEVATOR_LINUS                                        \
100    ((elevator_t) {                                               \
101        0,                         /* not used */                 \
102                                                                  \
103        1000000,                   /* read passovers */           \
104        2000000,                   /* write passovers */          \
105        0,                         /* max_bomb_segments */        \
106                                                                  \
107        0,                         /* not used */                 \
108        0,                         /* not used */                 \
109                                                                  \
110        elevator_linus,            /* elevator fn */              \
111        elevator_linus_merge,      /* elevator_merge_fn */        \
112        elevator_noop_dequeue,     /* dequeue_fn */               \
113        })
```

对块设备的数据结构初始化时，将其 request_queue_t 结构内的 elevator 设置成 ELEVATOR_LINUS，所以 765 行实际调用的是 elevator_linus_merge( )，其代码在 drivers/block/elevator.c 中：

[sys_read( ) > block_read( ) > ll_rw_block( ) > submit_bh( ) > generic_make_request( ) > __make_request( ) > elevator_linus_merge( )]

```
 53    int elevator_linus_merge(request_queue_t *q, struct request **req,
 54                    struct buffer_head *bh, int rw,
 55                    int *max_sectors, int *max_segments)
 56    {
 57        struct list_head *entry, *head = &q->queue_head;
 58        unsigned int count = bh->b_size >> 9, ret = ELEVATOR_NO_MERGE;
 59
 60        entry = head;
 61        if (q->head_active && !q->plugged)
 62            head = head->next;
 63
 64        while ((entry = entry->prev) != head) {
 65            struct request *__rq = *req = blkdev_entry_to_request(entry);
 66            if (__rq->sem)
 67                continue;
 68            if (__rq->cmd != rw)
 69                continue;
 70            if (__rq->nr_sectors + count > *max_sectors)
 71                continue;
 72            if (__rq->rq_dev != bh->b_rdev)
 73                continue;
 74            if (__rq->sector + __rq->nr_sectors == bh->b_rsector) {
 75                ret = ELEVATOR_BACK_MERGE;
```

```
76                  break;
77              }
78              if (!__rq->elevator_sequence)
79                  break;
80              if (__rq->sector - count == bh->b_rsector) {
81                  rq->elevator_sequence--;
82                  ret = ELEVATOR_FRONT_MERGE;
83                  break;
84              }
85          }
86
87          /*
88           * second pass scan of requests that got passed over, if any
89           */
90          if (ret != ELEVATOR_NO_MERGE && *req) {
91              while ((entry = entry->next) != &q->queue_head) {
92                  struct request *tmp = blkdev_entry_to_request(entry);
93                  tmp->elevator_sequence--;
94              }
95          }
96
97          return ret;
98      }
```

这里的 blkdev_entry_to_request( )是个宏操作，从一个 list_head 结构找到其宿主 request 结构，定义于 include/linux/blkdev.h：

```
188     #define blkdev_entry_to_request(entry) \
                            list_entry((entry), struct request, queue)
```

这个函数分两趟扫描当前的读写请求队列，返回对新来到的操作请求应如何与原有操作请求合并的指导(向前、向后、或不能合并)，并通过参数 req 返回可以与之合并的操作请求。为不至于过于冲淡操作的主线索，我们把这部分优化的详情留给有兴趣的读者。不过要指出，与新来到的操作请求合并以后可能会为进一步的合并创造条件，所以前面代码中的 777 和 791 行还要调用 attempt_back_merge( )或 attempt_front_merge( )作进一步的尝试。

如果合并成功，就不需要为新的读写请求单独创建一个数据结构并挂入队列了，否则就要继续往下跑，这就到了标号 get_rq 处。

[sys_read( ) > block_read( ) > ll_rw_block( ) > submit_bh( ) > generic_make_request( ) > __make_request( )]

```
804         /*
805          * Grab a free request from the freelist. Read first try their
806          * own queue - if that is empty, we steal from the write list.
807          * Writes must block if the write list is empty, and read aheads
808          * are not crucial.
809          */
```

```
810	get_rq:
811		if (freereq) {
812			req = freereq;
813			freereq = NULL;
814		} else if ((req = get_request(q, rw)) == NULL) {
815			spin_unlock_irq(&io_request_lock);
816			if (rw_ahead)
817				goto end_io;
818
819			freereq = __get_request_wait(q, rw);
820			goto again;
821		}
822
823	/* fill up the request-info, and add it to the queue */
824		req->cmd = rw;
825		req->errors = 0;
826		req->hard_sector = req->sector = sector;
827		req->hard_nr_sectors = req->nr_sectors = count;
828		req->current_nr_sectors = count;
829		req->nr_segments = 1; /* Always 1 for a new request. */
830		req->nr_hw_segments = 1; /* Always 1 for a new request. */
831		req->buffer = bh->b_data;
832		req->sem = NULL;
833		req->bh = bh;
834		req->bhtail = bh;
835		req->rq_dev = bh->b_rdev;
836		req->e = elevator;
837		add_request(q, req, head, latency);
838	out:
839		if (!q->plugged)
840			(q->request_fn)(q);
841		if (freereq)
842			blkdev_release_request(freereq);
843		spin_unlock_irq(&io_request_lock);
844		return 0;
845	end_io:
846		bh->b_end_io(bh, test_bit(BH_Uptodate, &bh->b_state));
847		return 0;
848	}
```

首先是分配一个 request 结构，这种数据结构是在 include/linux/blkdev.h 中定义的：

```
15	/*
16	 * Ok, this is an expanded form so that we can use the same
17	 * request for paging requests when that is implemented. In
18	 * paging, 'bh' is NULL, and the semaphore is used to wait
19	 * for read/write completion.
```

```
20     */
21    struct request {
22        struct list_head queue;
23        int elevator_sequence;
24        struct list_head table;
25
26        struct list_head *free_list;
27
28        volatile int rq_status; /* should split this into a few status bits */
29    #define RQ_INACTIVE              (-1)
30    #define RQ_ACTIVE                1
31    #define RQ_SCSI_BUSY             0xffff
32    #define RQ_SCSI_DONE             0xfffe
33    #define RQ_SCSI_DISCONNECTING    0xffe0
34
35        kdev_t rq_dev;
36        int cmd;        /* READ or WRITE */
37        int errors;
38        unsigned long sector;
39        unsigned long nr_sectors;
40        unsigned long hard_sector, hard_nr_sectors;
41        unsigned int nr_segments;
42        unsigned int nr_hw_segments;
43        unsigned long current_nr_sectors;
44        void * special;
45        char * buffer;
46        struct semaphore * sem;
47        struct buffer_head * bh;
48        struct buffer_head * bhtail;
49        request_queue_t *q;
50        elevator_t *e;
51    };
```

代码的作者在注释中说以后在页面换入/换出机制中也将使用这个数据结构。

内核中可供使用的 request 数据结构的数量是固定的，同时在调用 get_request( )时也说明了具体的操作（写操作比读操作慢，所以限制也更严）。如果分配失败，就说明系统中未完成的操作请求已经太多了。怎么办呢？那就要看具体情况了。如果所要求的只不过是预读，那本来就是可做可不做的，所以干脆就无功而返了。否则，要是非做不可的话，那就只好通过__get_request_wait( )睡眠等待了。在这个函数中还要针对任务队列 tq_disk 调用 run_task_queue( )，使所有可能尚未启动的操作请求队列得到启动，为 request 结构的回收创造条件。当 request 结构的回收使分配这种结构的要求得到满足时，睡眠中的进程就被唤醒而从__get_request_wait( )返回。显然，返回时指针 req 必定指向刚分配到的 request 结构而无需再加检验。不过，经过一段时间的睡眠，操作请求队列的情况可能已经变了（例如队列中的若干操作请求可能已经完成而不再在队列中了），所以要回到前面的标号 again 处，重新使指针 head 指向队列中的第一个请求。通常，当开始执行一个操作请求时要将其 request 结构从队列前端摘下，但是有些设备的驱动程序要到操作完成时才将它摘下，所以对这样的操作要跳过队列中的第一个 request

结构，而使指针 head 指向队列中的第二项。为了区分这两种不同的方式，在 request_queue_t 中设立了一个 head_active 字段，表示队列中的第一个 request 结构已经 active，因而应该把它跳过。不过，这只有在操作请求队列已经启动（不再在 tq_disk 队列中）时才需要(见 756~758 行)。当第二次到达 811 行时，freereq 一定已经指向一个空闲的 request 数据结构。

完成了对 request 结构的设置以后，就调用 add_request( )将其挂入设备的操作请求队列。这里的参数 latency 是在前面 744 行设置好了的，反映了当时对操作等待时间的预估值，用于优化目的，其实际上来自设备的 elevator_t 数据结构（request_queue_t 结构中的一部分）。

```
72    static inline int elevator_request_latency(elevator_t * elevator, int rw)
73    {
74        int latency;
75
76        latency = elevator->read_latency;
77        if (rw != READ)
78            latency = elevator->write_latency;
79
80        return latency;
81    }
```

函数 add_request( )的代码在 ll_rw_blk.c 中：

[sys_read( ) > block_read( ) > ll_rw_block( ) > submit_bh( ) > generic_make_request( )
 > __make_request( ) > add_request( )]

```
582    /*
583     * add-request adds a request to the linked list.
584     * It disables interrupts (acquires the request spinlock) so that it can muck
585     * with the request-lists in peace. Thus it should be called with no spinlocks
586     * held.
587     *
588     * By this point, req->cmd is always either READ/WRITE, never READA,
589     * which is important for drive_stat_acct( ) above.
590     */
591
592    static inline void add_request(request_queue_t * q, struct request * req,
593                                   struct list_head *head, int lat)
594    {
595        int major;
596
597        drive_stat_acct(req->rq_dev, req->cmd, req->nr_sectors, 1);
598
599        /*
600         * let selected elevator insert the request
601         */
602        q->elevator.elevator_fn(req, &q->elevator, &q->queue_head, head, lat);
603
```

```
604         /*
605         * FIXME(eric) I don't understand why there is a need for this
606         * special case code.  It clearly doesn't fit any more with
607         * the new queueing architecture, and it got added in 2.3.10.
608         * I am leaving this in here until I hear back from the COMPAQ
609         * people.
610             */
611         major = MAJOR(req->rq_dev);
612         if (major >= COMPAQ_SMART2_MAJOR+0 && major <= COMPAQ_SMART2_MAJOR+7)
613             (q->request_fn)(q);
614         if (major >= COMPAQ_CISS_MAJOR+0 && major <= COMPAQ_CISS_MAJOR+7)
615                 (q->request_fn)(q);
616         if (major >= DAC960_MAJOR+0 && major <= DAC960_MAJOR+7)
617             (q->request_fn)(q);
618     }
```

这里的 drive_stat_acct( )只是用来积累一些统计信息，我们不感兴趣。

如果队列原来是空的，那么很简单，通过 list_add( )把 request 数据结构挂入队列就是了，request 结构中的 list_head 结构就是为此目的而设的。如果队列中原来就有操作请求呢？显然，这意味着新的请求不能与已经存在的请求合并，所以应将其作为一个单独的请求挂入队列。如果不考虑进一步优化，把新的请求挂在队列的尾端也就可以了。但是，这里考虑了进一步的优化，即对操作路线的优化。所以通过 elevator 数据结构中的函数指针 elevator_fn 调用提供具体优化算法的函数，由这个函数来决定怎样将新的操作请求插入队列中。我们在前面已经看到其 elevator_fn 指针指向 elevator_linus( )，其代码在 drivers/block/elevator.c 中：

[sys_read( ) > block_read( ) > ll_rw_block( ) > submit_bh( ) > generic_make_request( ) > __make_request( ) > add_request( ) > elevator_linus( )]

```
29    /*
30     * Order ascending, but only allow a request to be skipped a certain
31     * number of times
32     */
33    void elevator_linus(struct request *req, elevator_t *elevator,
34                struct list_head *real head,
35                struct list_head *head, int orig latency)
36    {
37        struct list_head *entry = real_head;
38        struct request *tmp;
39
40        req->elevator_sequence = orig_latency;
41
42        while ((entry = entry->prev) != head) {
43            tmp = blkdev_entry_to_request(entry);
44            if (IN_ORDER(tmp, req))
45                break;
46            if (!tmp->elevator_sequence)
```

```
47                      break;
48                  tmp->elevator_sequence--;
49              }
50              list_add(&rcq->queue, entry);
51      }
```

这个函数所实现是对磁头移动路线的优化。设想在队列中已经存在两个操作请求，第一个是对第 100 号柱面上某几个扇区的操作，第二个是对第 800 号柱面上某几个扇区的操作。而新的请求是对第 150 号柱面上某几个扇区的操作。如果简单地把新的请求排在最后，那么在执行时先要把磁头移到第 100 号柱面，完成后再移到第 800 号柱面，最后又要回到第 150 号柱面。磁头的移动定位又是磁盘访问中最费时间的动作。这样的情况一多，磁头就疲于奔命而大大降低了效率。如果新的请求恰巧也在第 100 号柱面而本来可以不必移动磁头，那就更是冤枉了。可是，如果把新的请求插入到第 2 个请求之前，让磁头在完成了第一个请求的操作以后"顺路"先对第 150 号柱面操作，然后再到第 800 号柱面，那就可以显著改善效率了。所以，表面上是选择新请求在队列中的插入点，实质上却是对磁头移动的一种调度。代码中从队列的尾端开始向前扫描，以扇区号为依据（扇区号与柱面号是可以换算的），试图找到这么一个请求，即它所操作的扇区在新请求所操作的扇区之前，因而二者是"顺序"的。如果每次加入一个新操作请求时都保持顺序，那么整个队列就都是顺序的。宏操作 IN_ORDER 的定义在 include/linux/elevator.h 中：

```
61      /*
62       * This is used in the elevator algorithm.  We don't prioritise reads
63       * over writes any more --- although reads are more time-critical than
64       * writes, by treating them equally we increase filesystem throughput.
65       * This turns out to give better overall performance.  -- sct
66       */
67      #define IN_ORDER(s1, s2)                \
68          ((((s1)->rq_dev == (s2)->rq_dev &&   \
69              (s1)->sector < (s2)->sector)) || \
70           (s1)->rq_dev < (s2)->rq_dev)
```

这里把两个请求不在同一设备上、但是前者的次设备号小于后者的次设备号也考虑作顺序。但是，对于 IDE 接口，是每个设备一个队列（见 ide_get_queue( )的代码），所以这一点对 IDE 设备实际上不起作用。

如果找到了符合要求的操作请求，就把新请求插入到它的后面，从而实现了某种程度的优化。如果找不到，就把新请求排在队列的尾端，也就是说无法优化了。

我们在这里不对这些优化作定量的分析。一个优化算法应满足两个基本要求。一是有效性，即在多数可以优化的场合下真的得到了优化，提高了效率。二是公正性。以合并为例，如果我们每次都从队列的前端开始扫描，那么就有可能发生这样的情况：队列中原来就已经有很多操作请求在等待了，突然来了一大批新的操作请求，这些请求恰好都可以跟处于队列前端的请求合并，于是就全部"加塞"插到了大量原已在队列中等待的请求之前。这对于那些已经在等待的请求来说就不公平了。在极端的情况下，这些请求甚至可能"饿死"(starving)而一直得不到服务。作为队列，基本上的"先来先服务"原则还是应该遵循的。这就是为什么在优化和合并时都从队列尾端开始向前扫描的原因，并且即使从

队列尾端开始向前扫描也还应该对扫描的距离加以限制。最后，还有代价的考虑，优化的问题常常是NP 完全问题，所以在实际运用时都要“适而可止”，不能追求完美。

至此，将新的请求挂入操作请求队列的任务已经完成了，在 add_request( )中，还要检查一下操作的对象是否若干种设备之一(611～617 行)，如果是就要直接通过队列的函数指针 request_fn 启动队列中的第一个 I/O 操作。回到__make_request( )中，剩下的事情只有一件，那就是：如果操作请求队列的 plugged 标志为 0，也就是说尚未把队列插入到 tq_disk 中，此时就要直接通过队列的函数指针 request_fn 启动队列中的第一个 I/O 操作。正因为这样，代码的作者在__make_rrequest( )中加了注释，说既然有了后者就没有必要在 add_request( )中考虑那些特殊情况了。不过，在我们所讲的情景中，我们假定在__make_request( )中已经将它插入 bottom-half 的任务队列 tq_disk，所以 plugged 标志为 1。

这样，函数 ll_rw_block( )的执行就完成了，注意此时所要求的读/写实际上显然并未完成。实际的读/写是一个异步的过程，我们无法预测它究竟会在何时发生，因此需要使用或确认读/写结果的进程必须等待以取得同步。这通常是通过 wait_on_buffer( )完成的。这是一个 inline 函数，其代码在 include/linux/locks.h 中：

```
17    extern inline void wait_on_buffer(struct buffer_head * bh)
18    {
19        if (test_bit(BH_Lock, &bh->b_state))
20            __wait_on_buffer(bh);
21    }
```

函数__wait_on_buffer( )的代码在 fs/buffer.c 中：

```
136   /*
137    * Rewrote the wait-routines to use the "new" wait-queue functionality,
138    * and getting rid of the cli-sti pairs. The wait-queue routines still
139    * need cli-sti, but now it's just a couple of 386 instructions or so.
140    *
141    * Note that the real wait_on_buffer( ) is an inline function that checks
142    * if 'b_wait' is set before calling this, so that the queues aren't set
143    * up unnecessarily.
144    */
145   void __wait_on_buffer(struct buffer_head * bh)
146   {
147       struct task_struct *tsk = current;
148       DECLARE_WAITQUEUE(wait, tsk);
149
150       atomic_inc(&bh->b_count);
151       add_wait_queue(&bh->b_wait, &wait);
152       do {
153           run_task_queue(&tq_disk);
154           set_task_state(tsk, TASK_UNINTERRUPTIBLE);
155           if (!buffer_locked(bh))
156               break;
157           schedule( );
```

```
158         } while (buffer_locked(bh));
159         tsk->state = TASK_RUNNING;
160         remove_wait_queue(&bh->b_wait, &wait);
161         atomic_dec(&bh->b_count);
162     }
```

阅读了本书第 4 章和第 3 章的读者，对这段代码应该不会感到困难。等待着使用指定缓冲区（的内容）的进程在一个等待队列 wait 中睡眠，等待操作的完成。而对给定缓冲区的操作是否已经完成的标志，就是为了对缓冲区的操作所加的锁是否已被解除。所以，当给定缓冲区的 I/O 操作完成时，应该为该缓冲区解锁，并唤醒在该队列中睡眠的进程。但是，被唤醒并不一定意味着对特定缓冲区的操作已经完成，所以要在一个 do_while 循环中反复地测试和进入睡眠。另一方面，进程每次在 do_while 循环中进入睡眠前要通过 run_task_queue( )执行在 tq_disk 队列中的 bottom_half 函数，以确保对操作请求队列的执行已经启动。不过，对 run_task_queue( )的调用实际上可以来自好多不同的函数和场合，这里的调用并不是惟一的。

不管是从什么途径，当对 tq_disk 队列调用 run_task_queue( )时，就会依次把队列中的 tq_struct 数据结构从队列中解除并通过其函数指针 routine 调用相应的 bottom_half 函数。前面讲过，对于块设备操作请求队列，这个函数是 generic_unplug_device( )，现在我们来看它的代码，这是在 ll_rw_blk.c 中：

[run_task_queue( ) > __run_task_queue( ) > generic_unplug_device( )]

```
367     static void generic_unplug_device(void *data)
368     {
369         request_queue_t *q = (request_queue_t *) data;
370         unsigned long flags;
371 
372         spin_lock_irqsave(&io_request_lock, flags);
373         __generic_unplug_device(q);
374         spin_unlock_irqrestore(&io_request_lock, flags);
375     }
```

[run_task_queue( ) > __run_task_queue( ) > generic_unplug_device( ) > __generic_unplug_device( )]

```
355     /*
356      * remove the plug and let it rip..
357      */
358     static inline void __generic_unplug_device(request_queue_t *q)
359     {
360         if (q->plugged) {
361             q->plugged = 0;
362             if (!list_empty(&q->queue_head))
363                 q->request_fn(q);
364         }
365     }
```

在 tq_struct 结构中有一个 void 指针 data，用来传递对 bottom-half 函数的参数。对于操作请求队列

结构中的 tq_struct 结构 plug_tq，这个指针就指向其宿主 request_queue 数据结构。只要这个队列非空，这里就通过其函数指针 request_fn 调用该队列的 I/O 启动函数。这个启动函数是在块设备初始化时设置好的，对于作为主设备（primary）连接在 IDE 接口上的 IDE 硬盘，它是 do_ide_request( )，其代码在 drivers/ide/ide.c 中：

[run_task_queue( ) > __run_task_queue( ) > generic_unplug_device( ) > __generic_unplug_device( ) > do_ide_request( )]

```
1377    /*
1378     * Passes the stuff to ide_do_request
1379     */
1380    void do_ide_request(request_queue_t *q)
1381    {
1382        ide_do_request(q->queuedata, 0);
1383    }
```

操作请求队列中，即 request_queue_t 结构中，有个 void 指针 queuedata，可以根据具体设备的不同而指向不同的对象。对于 IDE 接口，这个指针指向一个 ide_hwgroup_t 数据结构，这也是在 ide.h 中定义的：

```
486     /*
487      * when ide_timer_expiry fires, invoke a handler of this type
488      * to decide what to do.
489      */
490     typedef int (ide_expiry_t)(ide_drive_t *);
491
492     typedef struct hwgroup_s {
493         ide_handler_t       *handler;/* irq handler, if active */
494         volatile int        busy;   /* BOOL: protects all fields below */
495         int                 sleeping; /* BOOL: wake us up on timer expiry */
496         ide_drive_t         *drive; /* current drive */
497         ide_hwif_t          *hwif;  /* ptr to current hwif in linked-list */
498         struct request      *rq;    /* current request */
499         struct timer_list   timer;  /* failsafe timer */
500         struct request      wrq;    /* local copy of current write rq */
501         unsigned long       poll_timeout;/* timeout value during long polls */
502         ide_expiry_t        *expiry;    /* queried upon timeouts */
503     } ide_hwgroup_t;
```

这个数据结构代表着一个“硬件组”，实际上是一组 IDE 接口，对属于同一组的 IDE 接口（以及接口上的磁盘）不能同时操作。不过，在大部分情况下每个 IDE 接口都能独立操作而互不影响，因而所谓一组 IDE 接口只包括一个 IDE 接口。IDE 接口是由 ide_hwif_t 数据结构代表的，内核中有个 ide_hwif_t 数组 ide_hwifs[ ]，以 IDE 接口的编号为下标。

```
191     ide_hwif_t      ide_hwifs[MAX_HWIFS];   /* master data repository */
```

这里的 MAX_HWIFS 定义为 6。一般的 PC 机有两个 IDE 硬盘接口，各自支持最多两个硬盘，此外还有一个 IDE 软盘接口，再要增加就得外加 IDE 接口卡了。数组 ide_hwifs[ ]中的每个元素都是一个 ide_hwif_t 数据结构，代表着系统中的一个 IDE 接口，定义于 include/linux/ide.h 中：

```
421     typedef struct hwif_s {
422         struct hwif_s   *next;              /* for linked-list in ide_hwgroup_t */
423         void            *hwgroup;           /* actually (ide_hwgroup_t *) */
424         ide_ioreg_t io_ports[IDE_NR_PORTS]; /* task file registers */
425         hw_regs_t   hw;                     /* Hardware info */
426         ide_drive_t drives[MAX_DRIVES];     /* drive info */
427         struct gendisk  *gd;                /* gendisk structure */
428         ide_tuneproc_t  *tuneproc;          /* routine to tune PIO mode for drives */
429         ide_speedproc_t *speedproc;     /* routine to retune DMA modes for drives */
430         ide_selectproc_t *selectproc;       /* tweaks hardware to select drive */
431         ide_resetproc_t *resetproc;/* routine to reset controller after a disk reset */
432         ide_intrproc_t  *intrproc;
                        /* special interrupt handling for shared pci interrupts */
433         ide_maskproc_t  *maskproc;  /* special host masking for drive selection */
434         ide_quirkproc_t *quirkproc; /* check host's drive quirk list */
435         ide_rw_proc_t   *rwproc;    /* adjust timing based upon rq->cmd direction */
436         ide_dmaproc_t   *dmaproc;   /* dma read/write/abort routine */
437         unsigned int    *dmatable_cpu;
                                /* dma physical region descriptor table (cpu view) */
438         dma_addr_t  dmatable_dma; /* dma physical region descriptor table (dma view) */
439         struct scatterlist *sg_table; /* Scatter-gather list used to build the above */
440         int sg_nents;                   /* Current number of entries in it */
441         int sg_dma_direction;           /* dma transfer direction */
442         struct hwif_s   *mate;          /* other hwif from same PCI chip */
443         unsigned long   dma_base;       /* base addr for dma ports */
444         unsigned        dma_extra;      /* extra addr for dma ports */
445         unsigned long   config_data;    /* for use by chipset-specific code */
446         unsigned long   select_data;    /* for use by chipset-specific code */
447         struct proc_dir_entry *proc;    /* /proc/ide/ directory entry */
448         int     irq;                    /* our irq number */
449         byte        major;              /* our major number */
450         char        name[6];            /* name of interface, eg. "ide0" */
451         byte        index;              /* 0 for ide0; 1 for ide1; ... */
452         hwif_chipset_t  chipset;        /* sub-module for tuning.. */
453         unsigned    noprobe    : 1;     /* don't probe for this interface */
454         unsigned    present    : 1;     /* this interface exists */
455         unsigned    serialized : 1;     /* serialized operation with mate hwif */
456         unsigned    sharing_irq: 1;     /* 1 = sharing irq with another hwif */
457         unsigned    reset      : 1;     /* reset after probe */
458         unsigned    autodma    : 1;     /* automatically try to enable DMA at boot */
459         unsigned    udma_four  : 1;     /* 1=ATA-66 capable, 0=default */
460         byte    channel;            /* for dual-port chips: 0=primary, 1=secondary */
461     #ifdef CONFIG_BLK_DEV_IDEPCI
```

```
462         struct pci_dev  *pci_dev;           /* for pci chipsets */
463         ide_pci_devid_t pci_devid;          /* for pci chipsets: {VID,DID} */
464     #endif                                  /* CONFIG_BLK_DEV_IDEPCI */
465     #if (DISK_RECOVERY_TIME > 0)
466         unsigned long   last_time;          /* time when previous rq was done */
467     #endif
468         byte        straight8;              /* Alan's straight 8 check */
469         void        *hwif_data;             /* extra hwif data */
470     } ide_hwif_t;
```

结构中的数组 drives[ ]对应着可以连接到同一个 IDE 接口上的若干个硬盘，常数 MAX_DRIVES 定义为 2，所以数组的大小为 2。指针 hwgroup 就指向一个 ide_hwgroup_t 结构，

通常，每个 IDE 接口都有一个 ide_hwgroup_t 数据结构，二者以它们的指针 hwgroup 和 hwif 互相指向对方。同一个 IDE 接口上的两个磁盘不能同时操作，所以 ide_hwgroup_t 结构代表着一组不能同时操作而必须把对它们的操作“串行化”的磁盘。但是，也有些 IDE 接口卡把两个 IDE 接口做在一起而不能同时操作，于是就共用一个 ide_hwgroup_t 数据结构，此时两个 ide_hwif_t 结构就要通过它们的指针 next 链接在一起，而 ide_hwgroup_t 结构中的 hwif 指针则指向当前 IDE 接口。所以 ide_hwgroup_t 是比 ide_hwif_t 更高一层的数据结构。也就是说：

- 每个 ide_hwgroup_t 数据结构代表着一组不能同时操作的 IDE 接口。
- 每个 ide_hwif_t 数据结构代表着一个 IDE 接口，也就是一组不能同时操作的 IDE 设备，ide_hwif_t 结构中有个 ide_drive_t 结构数组。
- 每个 ide_drive_t 数据结构代表着一个 IDE 设备。

在 PC 机中，通常一个“硬件组”只包含一个 IDE 接口，而一个 IDE 接口包含一个或两个磁盘。

每个 IDE 设备，即 ide_drive_t 数据结构，都有自己的操作请求队列，但是整个 IDE 接口组只能有一个“当前操作请求”，即正在为之服务的操作请求，ide_hwgroup_t 结构中的指针 rq 就指向这个请求的 request 数据结构。

回过来看 ide_do_request( )的代码，这是在 ide.c 中，代码的作者在函数前面加了很长一段注释，读者不妨先读一下：

```
1254    /*
1255     * Issue a new request to a drive from hwgroup
1256     * Caller must have already done spin_lock_irqsave(&io_request_lock, ..);
1257     *
1258     * A hwgroup is a serialized group of IDE interfaces.  Usually there is
1259     * exactly one hwif (interface) per hwgroup, but buggy controllers (eg. CMD640)
1260     * may have both interfaces in a single hwgroup to "serialize" access.
1260     * Or possibly multiple ISA interfaces can share a common IRQ by being grouped
1262     * together into one hwgroup for serialized access.
1263     *
1264     * Note also that several hwgroups can end up sharing a single IRQ,
1265     * possibly along with many other devices.  This is especially common in
1266     * PCI-based systems with off-board IDE controller cards.
1267     *
```

```
1268     * The IDE driver uses the single global io_request_lock spinlock to protect
1269     * access to the request queues, and to protect the hwgroup->busy flag.
1270     *
1271     * The first thread into the driver for a particular hwgroup sets the
1272     * hwgroup->busy flag to indicate that this hwgroup is now active,
1273     * and then initiates processing of the top request from the request queue.
1274     *
1275     * Other threads attempting entry notice the busy setting, and will simply
1276     * queue their new requests and exit immediately.  Note that hwgroup->busy
1277     * remains set even when the driver is merely awaiting the next interrupt.
1278     * Thus, the meaning is "this hwgroup is busy processing a request".
1279     *
1280     * When processing of a request completes, the completing thread or IRQ-handler
1281     * will start the next request from the queue.  If no more work remains,
1282     * the driver will clear the hwgroup->busy flag and exit.
1283     *
1284     * The io_request_lock (spinlock) is used to protect all access to the
1285     * hwgroup->busy flag, but is otherwise not needed for most processing in
1286     * the driver.  This makes the driver much more friendlier to shared IRQs
1287     * than previous designs, while remaining 100% (?) SMP safe and capable.
1288     */
```

读者可以先大致看一下，然后在阅读有关的代码时再回过来细读。下面是函数的代码：

[run_task_queue( ) > __run_task_queue( ) > generic_unplug_device( ) > __generic_unplug_device( ) > do_ide_request( ) > ide_do_request( )]

```
1289    static void ide_do_request(ide_hwgroup_t *hwgroup, int masked_irq)
1290    {
1291        ide_drive_t *drive;
1292        ide_hwif_t  *hwif;
1293        ide_startstop_t startstop;
1294
1295        ide_get_lock(&ide_lock, ide_intr, hwgroup);
                            /* for atari only: POSSIBLY BROKEN HERE(?) */
1296
1297          cli( ); /* necessary paranoia: ensure IRQs are masked on local CPU */
1298
1299        while (!hwgroup->busy) {
1300            hwgroup->busy = 1;
1301            drive = choose_drive(hwgroup);
1302            if (drive == NULL) {
1303                unsigned long sleep = 0;
1304                hwgroup->rq = NULL;
1305                drive = hwgroup->drive;
1306                do {
1307                    if (drive->sleep &&
                            (!sleep || 0 < (signed long)(sleep - drive->sleep)))
```

```
1308                    sleep = drive->sleep;
1309                } while ((drive = drive->next) != hwgroup->drive);
1310                if (sleep) {
1311                    /*
1312                     * Take a short snooze, and then wake up this hwgroup again.
1313                     * This gives other hwgroups on the same a chance to
1314                     * play fairly with us, just in case there are big differences
1315                     * in relative throughputs.. don't want to hog the cpu too much.
1316                     */
1317                    if (0 < (signed long)(jiffies + WAIT_MIN_SLEEP - sleep))
1318                        sleep = jiffies + WAIT_MIN_SLEEP;
1319    #if 1
1320                    if (timer_pending(&hwgroup->timer))
1321                        printk("ide_set_handler: timer already active\n");
1322    #endif
1323                    hwgroup->sleeping = 1;
                                /* so that ide_timer_expiry knows what to do */
1324                    mod_timer(&hwgroup->timer, sleep);
1325                    /* we purposely leave hwgroup->busy==1 while sleeping */
1326                } else {
1327                    /* Ugly, but how can we sleep for the lock otherwise?
                            perhaps from tq_disk? */
1328                    ide_release_lock(&ide_lock);    /* for atari only */
1329                    hwgroup->busy = 0;
1330                }
1331                return;      /* no more work for this hwgroup (for now) */
1332            }
1333            hwif = HWIF(drive);
1334            if (hwgroup->hwif->sharing_irq && hwif != hwgroup->hwif &&
                                        hwif->io_ports[IDE_CONTROL_OFFSET]) {
1335                /* set nIEN for previous hwif */
1336                SELECT_INTERRUPT(hwif, drive);
1337            }
1338            hwgroup->hwif = hwif;
1339            hwgroup->drive = drive;
1340            drive->sleep = 0;
1341            drive->service_start = jiffies;
1342
1343            if ( drive->queue.plugged ) /* paranoia */
1344                printk("%s: Huh? nuking plugged queue\n", drive->name);
1345            hwgroup->rq = blkdev_entry_next_request(&drive->queue.queue_head);
1346            /*
1347             * Some systems have trouble with IDE IRQs arriving while
1348             * the driver is still setting things up.  So, here we disable
1349             * the IRQ used by this interface while the request is being started.
1350             * This may look bad at first, but pretty much the same thing
1351             * happens anyway when any interrupt comes in, IDE or otherwise
1352             *  -- the kernel masks the IRQ while it is being handled.
```

```
1353            */
1354            if (masked_irq && hwif->irq != masked_irq)
1355                disable_irq_nosync(hwif->irq);
1356            spin unlock(&io_request lock);
1357            ide__sti( );    /* allow other IRQs while we start this request */
1358            startstop = start_request(drive);
1359            spin_lock_irq(&io_request_lock);
1360            if (masked irq && hwif->irq != masked_irq)
1361                enable_irq(hwif->irq);
1362            if (startstop == ide_stopped)
1363                hwgroup->busy = 0;
1364        }
1365    }
```

对于 i386 结构的 CPU，ide_get_lock( )只是一个空函数。同时，对设备的操作必须是独占的，绝不容许受到干扰。所以这里采取了最为严厉的措施，即通过__cli( )把中断关掉，这样就既不会有中断发生，也不会有进程调度发生了。那么，在 SMP 多处理器结构的系统中来自其他 CPU 的干扰呢？那也已经考虑到了，请回过去看 generic_unplug_device( )的代码，通过函数指针 request_fn 对 do_ide_request( )的调用是在由 spin_lock_irqsave( )和 spin_unlock_irqrestore( )围成的临界区中进行的，一个 CPU 进入了这个区域就把大门锁上了，别的 CPU 想要进来就只好在 spin_lock_irqsave( )中等待。

然后，只要 IDE 接口不是正在忙，就可以进入 while 循环的循环体内，并且马上就把这个 IDE 接口的 busy 标志设成 1，又关上了一道门。这道门一关上，即使把外层的大门打开，例如把中断打开，也不会有其他进程（或中断服务程序）能进入这个 while 循环了。不过，现在还不能把中断打开，因为中断服务程序有可能不经由这里的 while 循环而直接访问 IDE 接口造成干扰。

由于同一个 IDE 接口上可能有不止一个磁盘，这时候就要确定接下去到底是对哪一个磁盘操作了。这就是调用 choose_drive( )的目的。如前所述，在 ide_hwif_t 数据结构中有个 ide_drive_t 结构数组 drives[ ]，而相应的 ide_hwgroup_t 结构中则有一个 ide_drive_t 结构指针 drive，它指向上述 drives[ ]数组中的一个数据结构，这就是整个组的当前磁盘。不过，这个指针所指向的实际上是个环形链，因为存在于同一 IDE 接口上的所有磁盘所对应的 ide_drive_t 数据结构都通过结构中的 next 指针连成一个环形链。所以 choose_drive( )的任务就是扫描这个环形链，察看每个 ide_drive_t 结构中的操作请求队列，以确定哪一个磁盘应该是下一步的操作对象。

此处先请读者暂时停一下，先将上面提到的有关 IDE 硬盘驱动的几个数据结构以及相互之间的关系总结一下，并画出联系图。这对程序理解会有好处。

前面说过，同一个“硬件组”，即共用同一个 ide_hwgroup_t 结构的硬盘（通常在同一个 IDE 接口上）是不能同时操作的。但是，这句话不够确切。更确切地应该说是不能同时启动操作，或者说不能同时向两个硬盘发出操作命令，两个磁盘的物理操作（例如移动磁头然后读出）完全可以并行。当向一个硬盘发出操作命令启动其操作的时候，就在相应 ide_drive_t 数据结构里的 service_start 字段中记下当时的时间，根据这个起始时间以及具体操作的大小就可以估计出这个操作完成的时间。另一方面，每当一个操作完成（或因超时而失败）时，可以根据当时的时间和操作的启动时间计算出本次操作实际耗用的时间。这个时间记录在相应 ide_drive_t 结构中的 sevice_time 字段中，可以用作具体硬盘操作速度的一个参考值。在某些场合下，还可以在自愿的基础上（ide_drive_t 结构中的 nice1 标志为 1）将

一个硬盘投入睡眠状态，也就是在预定的某个时间之前不对其进行操作，这个预定的时间就记录在 ide_drive_t 结构里的 sleep 字段中。不过要注意，这里所说的是在一定的时间内不对其进行操作，也就是不向其发出操作命令，但是这并不意味着在这段时间内它就一定是在空转，只不过是估计在此之前正在进行中的操作肯定不会完成而已。在挑选下一个操作对象时，这些因素都应该考虑进去。函数 choose_drive( )的代码在 ide.c 中，我们就把它留给读者了。

挑选下一个操作对象的结果有可能是空。原因可能有二，一是所有磁盘的操作请求队列都是空，因而无事可干；二是所有的磁盘都在睡眠状态，不能向它们发出操作命令。所以，代码中通过一个 do_while 循环扫描 ide_drive_t 结构的环形链，找出其中最早的睡眠结束时间。如果这个最早的结束时间非 0，那就说明找不出一个操作对象的原因确是因睡眠引起的，因此将 ide_hwgroup_t 结构中的 sleeping 标志设成 1，表示整个磁盘组都在睡眠(1323 行)；同时设置一个定时器，让它在到点的时候将该 ide_hwgroup_t 结构“唤醒”。否则，如果最早的睡眠结束时间为 0，就说明该组磁盘的操作请求队列全是空的，所以把它的 busy 标志设成 0。既然没有下一个操作对象，本次 ide_do_reguest( )就结束了（见 1331 行）。

如果找到了下一个操作对象(见 1333 行)，那就先找到该操作对象，即磁盘所属的 IDE 接口，这里宏操作 HWIF 的定义见 include/linux/ide.h：

```
96      #define HWIF(drive)        ((ide_hwif_t *)((drive)->hwif))
```

读者也许感到奇怪，在 ide_hwgroup_t 结构中不是有个指针 hwif 指向当前 ide_hwif_t 数据结构吗？是的，但是如果有两个 IDE 接口共用同一个 ide_hwgroup_t 结构的话，新挑选的 ide_drive_t 结构可能不属于原先的“当前 IDE 接口”。如果真是这样的话（见 1334 行），那就意味着 IDE 接口的切换，这时候就要先向原 IDE 接口的控制寄存器（如果存在的话，见 1334 行）写一个控制字节将该接口的中断请求关闭。这里的宏操作 SELECT_INTERRUPT 定义于 include/linux/ide.h：

```
189     #define SELECT_INTERRUPT(hwif, drive)                  \
190     {                                         \
191          if (hwif->intrproc)                       \
192              hwif->intrproc(drive);                    \
193          else                                 \
194              OUT_BYTE((drive)->ctl|2, hwif->io_ports[IDE_CONTROL_OFFSET]);\
195     }
```

而宏操作 OUT_BYTE 的第 1 个参数为需要写出的字节，当这个字节的 bit1 为 1 时，该 IDE 接口的中断请求就被屏蔽掉了。第 2 个参数为寄存器的 I/O 地址，代表着具体 IDE 接口的 ide_hwif_t 结构中有个数组 io_ports[ ]，该数组提供了接口上各个寄存器的 I/O 地址。

然后，就可以使 ide_hwgroup_t 结构中的指针 hwif 和 drive 指向新的当前 IDE 接口和磁盘了。全局量 jiffies 代表当前时间（见第 3 章），这就是本次操作的启动时间 sevice_start。代码中还对操作对象的操作请求队列加以检验，确保它已经不在 tq_disk 队列中。

接着，使指针 hwgroup->rq 指向操作请求队列中的第一个操作请求（但不将其从队列中脱链），这就是当前操作请求。

至此，整个系统的中断仍是关着的，但是马上就要把它打开了（见 1357 行），因为为了一个具体

的设备而长时间关闭全系统的中断机制是不合适的。可是，对当前磁盘的操作仍不允许打扰，所以，在打开全系统的中断之前可能需要先通过 disable_irg_nosync( )将具体的中断服务程序关闭，也就是说在打开大门之前先把里面具体房间的门关上。不过，这也是有条件的，这里 masked_irg 是作为参数从 do_ide_request( )传下来的，从它的代码中可以看到这个参数是 0，所以这里 1354 行 if 语句的条件不会得到满足，也就是说不必先把里面具体房间的门关上。如前所述，在我们这条执行路线中实际上已经关上了一道门，那就是 IDE 接口的 busy 标志。

做好了这些准备，现在可以根据具体的操作请求向目标磁盘发出操作命令了，这是由 start_request( )完成的，其代码在 ide.c 中。读者将会看到，这是一个很重要的函数。

[run_task_queue( ) > __run_task_queue( ) > generic_unplug_device( ) > __generic_unplug_device( ) > do_ide_request( ) > ide_do_request( ) > start_request( )]

```
1129     /*
1130      * start_request( ) initiates handling of a new I/O request
1131      */
1132     static ide_startstop_t start_request (ide_drive_t *drive)
1133     {
1134         ide_startstop_t startstop;
1135         unsigned long block, blockend;
1136         struct request *rq = blkdev_entry_next_request(&drive->queue.queue_head);
1137         unsigned int minor = MINOR(rq->rq_dev), unit = minor >> PARTN_BITS;
1138         ide_hwif_t *hwif = HWIF(drive);
1139
1140     #ifdef DEBUG
1141         printk("%s: start_request: current=0x%08lx\n", hwif->name, (unsigned long) rq);
1142     #endif
1143         if (unit >= MAX_DRIVES) {
1144             printk("%s: bad device number: %s\n", hwif->name, kdevname(rq->rq_dev));
1145             goto kill_rq;
1146         }
1147     #ifdef DEBUG
1148         if (rq->bh && !buffer_locked(rq->bh)) {
1149             printk("%s: block not locked\n", drive->name);
1150             goto kill_rq;
1151         }
1152     #endif
1153         block    = rq->sector;
1154         blockend = block + rq->nr_sectors;
1155
1156         if ((rq->cmd == READ || rq->cmd == WRITE) &&
1157             (drive->media == ide_disk || drive->media == ide_floppy)) {
1158             if ((blockend < block) ||
                         (blockend > drive->part[minor&PARTN_MASK].nr_sects)) {
1159             printk("%s%c: bad access: block=%ld, count=%ld\n", drive->name,
1160             (minor&PARTN_MASK)?'0'+(minor&PARTN_MASK):' ', block, rq->nr_sectors);
1161                 goto kill_rq;
```

```
1162            }
1163            block += drive->part[minor&PARTN_MASK].start_sect + drive->sect0;
1164        }
1165        /* Yecch - this will shift the entire interval,
1166           possibly killing some innocent following sector */
1167        if (block == 0 && drive->remap_0_to_1 == 1)
1168            block = 1;  /* redirect MBR access to EZ-Drive partn table */
1169
1170    #if (DISK_RECOVERY_TIME > 0)
1171        while ((read_timer( ) - hwif->last_time) < DISK_RECOVERY_TIME);
1172    #endif
1173
1174        SELECT_DRIVE(hwif, drive);
1175        if (ide_wait_stat(&startstop, drive, drive->ready_stat,
                                        BUSY_STAT|DRQ_STAT, WAIT_READY)) {
1176            printk("%s: drive not ready for command\n", drive->name);
1177            return startstop;
1178        }
1179        if (!drive->special.all) {
1180            if (rq->cmd == IDE_DRIVE_CMD || rq->cmd == IDE_DRIVE_TASK) {
1181                return execute_drive_cmd(drive, rq);
1182            }
1183            if (drive->driver != NULL) {
1184                return (DRIVER(drive)->do_request(drive, rq, block));
1185            }
1186            printk("%s: media type %d not supported\n", drive->name, drive->media);
1187            goto kill_rq;
1188        }
1189        return do_special(drive);
1190    kill_rq:
1191        if (drive->driver != NULL)
1192            DRIVER(drive)->end_request(0, HWGROUP(drive));
1193        else
1194            ide_end_request(0, HWGROUP(drive));
1195        return ide_stopped;
1196    }
```

首先是对参数的检查。这里的局部量 block 和 blockend 代表着操作的起始扇区号和结束扇区号，而不是逻辑块号。值得注意的是 1163 行，如果所操作的"磁盘"实际上只是磁盘上的一个分区，那就要将逻辑磁盘的扇区号映射成物理磁盘上的扇区号。在物理磁盘划分成多个分区的情况下，相应 ide_drive_t 结构中的数组 part[ ]提供了各个分区的参数，包括各个分区的起始逻辑扇区号。这个扇区号之所以仍旧是"逻辑扇区号"，是因为磁盘上存在着一个"分区表"，这个分区表使逻辑上的扇区 0 移到了另一个物理位置上，这个位置就是 drive->sect0。此外，在有磁盘分区存在的情况下，有可能要将用于引导块的扇区 0 映射到扇区 1 上，此时 ide_drive_t 结构中的 remap0_to_1 标志为 1（见 1167 行）。这些信息都是在块设备初始化时设置好了的。

有些磁盘在相继的两次操作之间要有个"恢复时间"，对这样的磁盘要通过 1171 行的 while 循环保

证其恢复时间，但是大多数磁盘都没有这个要求。

下面的 SELECT_DRIVE 是个宏操作，与 SELECT_INTERRUPT 相似，其定义见 ide.h:

```
182     #define SELECT_DRIVE(hwif,drive)                    \
183     {                                           \
184         if (hwif->selectproc)                   \
185             hwif->selectproc(drive);            \
186         OUT_BYTE((drive)->select.all, hwif->io_ports[IDE_SELECT_OFFSET]); \
187     }
```

有些 IDE 设备要求在写入其“驱动器/磁头选择寄存器”之前先执行一些特殊的处理，所以这样的设备可以将相应 ide_drive_t 结构中的函数指针 selectproc 预先设置好，这里就可以通过函数指针调用这个函数了。但是一般 IDE 磁盘都没有这个要求。这里主要是设置硬盘的“驱动器/磁头选择寄存器”，这个 8 位寄存器的格式如图 8.6 所示。

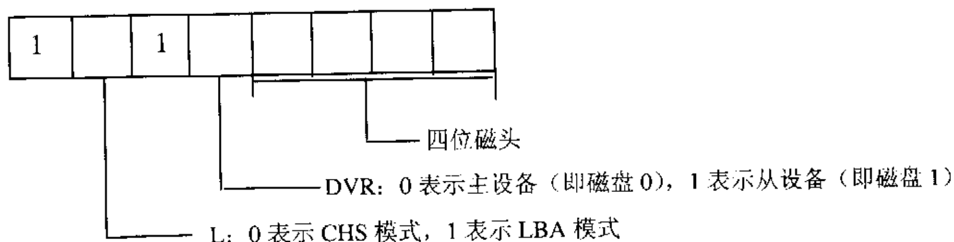


图 8.6　驱动器/磁头选择寄存器格式定义

相应的数据结构定义于 include/asm_i386/ide.h 中：

```
89    typedef union {
90        unsigned all            : 8;    /* all of the bits together */
91        struct {
92            unsigned head       : 4;    /* always zeros here */
93            unsigned unit       : 1;    /* drive select number, 0 or 1 */
94            unsigned bit5       : 1;    /* always 1 */
95            unsigned lba        : 1;    /* using LBA instead of CHS */
96            unsigned bit7       : 1;    /* always 1 */
97        } b;
98    } select_t;
```

对 IDE 磁盘上的扇区有两种不同的寻址方式。第一种是所谓 CHS（柱面/磁头/扇区）模式。由于“驱动器/磁头选择寄存器”中只有 4 位用于磁头选择，最多就只能有 16 个磁头。同时，另外两个用于柱面号的寄存器又将柱面数限制到 1024，而扇区号寄存器又将(每磁道)扇区的数量限制到 256。可是，要把一个圆(磁道)划分成 256 个扇区是有困难的，所以实际上只能划分成 64 个扇区。这么一来，扇区的总量就限制为 1024×16×64=1 M，由于每个扇区的容量为 512 字节，IDE 磁盘的总容量就被限制为 0.5 GB。为了克服这个缺点，在“扩充 IDE”的标准中又提供了一种 LBA（逻辑块号）模式，将整个

磁盘看作连续的逻辑扇区阵列，而由 IDE 磁盘（更确切地说是 EIDE 磁盘）自己进行从逻辑扇区号到柱面、磁头、扇区的转换，这样，由原来两个用于柱面号的寄存器，加上原来用于扇区号的寄存器，再加上原来用于磁头号的 4 位，一共有 28 位用于逻辑区号，使理论上的最大容量达到了 128 GB。将驱动器/磁头选择寄存器中的 bit6 设置成 1，就使磁盘工作于 LBA 模式。

有些 IDE 设备有些特殊操作，例如在每次读/写以后都要让磁盘回到 0 号柱面等等。特殊操作带有初始化的性质而只需执行一次。为了照顾到这些有特殊要求的磁盘，ide_drive_t 结构中设置了一个字段 special，它的类型是 special_t，定义见 include/linux/ide.h：

```
268    typedef union {
269        unsigned all                 : 8;    /* all of the bits together */
270        struct {
271            unsigned set_geometry    : 1;    /* respecify drive geometry */
272            unsigned recalibrate     : 1;    /* seek to cyl 0        */
273            unsigned set_multmode    : 1;    /* set multmode count */
274            unsigned set_tune        : 1;    /* tune interface for drive */
275            unsigned reserved        : 4;    /* unused */
276        } b;
277    } special_t;
```

块设备初始化时，如果具体的磁盘有特殊要求，就相应地设置其 ide_drive_t 结构中这个字段的值，否则设置成 0。

先看没有特殊要求的情景（见 1179 行），这是主流。对 IDE 磁盘的操作可以分成两类。一类是起控制作用的命令，如单纯的磁头定位、自检、光盘驱动器的关门/开门等。这一类的命令由 execute_drive_cmd()启动，这些操作请求通常不是由用户进程（应用程序）发出的。另一类就是我们所关心的数据读/写命令，通过具体设备的驱动函数跳转表，即 ide_driver_t 结构中的函数指针 do_request 启动，宏操作 DRIVER 的定义见 include/linux/ide.h：

```
620    #define DRIVER(drive)         ((ide_driver_t *)((drive)->driver))
```

如前所述，ide_drive_t 数据结构中有个指针 driver，指向一个 ide_driver_t 数据结构（注意不要混淆这两种不同的数据结构），这就是具体设备的驱动函数跳转表。例如，一般 IDE 硬盘的函数表为 idedisk_driver，光盘驱动器的函数表为 ide_cdrom_driver，软盘的函数表为 idefloppy_driver，IDE 磁带驱动器的函数表为 idetape_driver。显然，这种数据结构起着类似于 file_operations 数据结构的作用，其类型定义见 ide.h：

```
599    typedef struct ide_driver_s {
600        const char            *name;
601        const char            *version;
602        byte                  media;
603        unsigned busy                 : 1;
604        unsigned supports_dma         : 1;
605        unsigned supports_dsc_overlap    : 1;
606        ide_cleanup_proc          *cleanup;
```

```
607         ide_do_request_proc      *do_request;
608         ide_end_request_proc     *end_request;
609         ide_ioctl_proc           *ioctl;
610         ide_open_proc            *open;
611         ide_release_proc         *release;
612         ide_check_media_change_proc *media_change;
613         ide_revalidate_proc      *revalidate;
614         ide_pre_reset_proc       *pre_reset;
615         ide_capacity_proc        *capacity;
616         ide_special_proc         *special;
617         ide_proc_entry_t         *proc;
618     } ide_driver_t;
```

数据结构 idedisk_driver 的值定义于 ide.c 中，为阅读方便再次把它列出如下：

```
711     /*
712      *  IDE subdriver functions, registered with ide.c
713      */
714     static ide_driver_t idedisk_driver = {
715         "ide-disk",              /* name */
716         IDEDISK_VERSION,         /* version */
717         ide_disk,                /* media */
718         0,                       /* busy */
719         1,                       /* supports_dma */
720         0,                       /* supports_dsc_overlap */
721         NULL,                    /* cleanup */
722         do_rw_disk,              /* do_request */
723         NULL,                    /* end_request */
724         NULL,                    /* ioctl */
725         idedisk_open,            /* open */
726         idedisk_release,         /* release */
727         idedisk_media_change,    /* media_change */
728         idedisk_revalidate,      /* revalidate */
729         idedisk_pre_reset,       /* pre_reset */
730         idedisk_capacity,        /* capacity */
731         idedisk_special,         /* special */
732         idedisk_proc             /* proc */
733     };
```

可见，其函数指针 do_request 指向 do_rw_disk( )，这个函数的代码也在 ide_disk.c 中。这又是一个很重要的函数，由于代码比较长，我们分段阅读。

[run_task_queue( ) > __run_task_queue( ) > generic_unplug_device( ) > __generic_unplug_device( ) > do_ide_request( ) > ide_do_request( ) > start_request( ) > do_rw_disk( )]

```
377     /*
378      * do_rw_disk( ) issues READ and WRITE commands to a disk,
```

```
379     * using LBA if supported, or CHS otherwise, to address sectors.
380     * It also takes care of issuing special DRIVE_CMDs.
381     */
382    static ide_startstop_t do_rw_disk (ide_drive_t *drive,
                                          struct request *rq, unsigned long block)
383    {
384        if (IDE_CONTROL_REG)
385            OUT_BYTE(drive->ctl, IDE_CONTROL_REG);
386        OUT_BYTE(rq->nr_sectors, IDE_NSECTOR_REG);
387    #ifdef CONFIG_BLK_DEV_PDC4030
388        if (drive->select.b.lba || IS_PDC4030_DRIVE) {
389    #else /* !CONFIG_BLK_DEV_PDC4030 */
390        if (drive->select.b.lba) {
391    #endif /* CONFIG_BLK_DEV_PDC4030 */
392    #ifdef DEBUG
393            printk("%s: %sing: LBAsect=%ld, sectors=%ld, buffer=0x%08lx\n",
394                drive->name, (rq->cmd==READ)?"read":"writ",
395                block, rq->nr_sectors, (unsigned long) rq->buffer);
396    #endif
397            OUT_BYTE(block, IDE_SECTOR_REG);
398            OUT_BYTE(block>>=8, IDE_LCYL_REG);
399            OUT_BYTE(block>>=8, IDE_HCYL_REG);
400            OUT_BYTE(((block>>8)&0x0f)|drive->select.all, IDE_SELECT_REG);
401        } else {
402            unsigned int sect, head, cyl, track;
403            track = block / drive->sect;
404            sect  = block % drive->sect + 1;
405            OUT_BYTE(sect, IDE_SECTOR_REG);
406            head  = track % drive->head;
407            cyl   = track / drive->head;
408            OUT_BYTE(cyl, IDE_LCYL_REG);
409            OUT_BYTE(cyl>>8, IDE_HCYL_REG);
410            OUT_BYTE(head|drive->select.all, IDE_SELECT_REG);
411    #ifdef DEBUG
412            printk("%s: %sing: CHS=%d/%d/%d, sectors=%ld, buffer=0x%08lx\n",
413                drive->name, (rq->cmd==READ)?"read":"writ", cyl,
414                head, sect, rq->nr_sectors, (unsigned long) rq->buffer);
415    #endif
416        }
417    #ifdef CONFIG_BLK_DEV_PDC4030
418        if (IS_PDC4030_DRIVE) {
419            extern ide_startstop_t do_pdc4030_io(ide_drive_t *, struct request *);
420            return do_pdc4030_io (drive, rq);
421        }
422    #endif /* CONFIG_BLK_DEV_PDC4030 */
```

这里的 IDE_CONTROL_REG 定义见 ide.h:

```
126     #define IDE_CONTROL_REG       (HWIF(drive)->io_ports[IDE_CONTROL_OFFSET])
```

其他一些寄存器 I/O 地址的定义也与此相似。与前面 ide_do_request( )代码中 1336 行调用的 SELECT_INTERRUPT 作一对比，就可以看出这一次的 bit1 为 0，表示解除对 IDE 接口中断请求的屏蔽。然后把需要读/写的扇区数量写入磁盘的“扇区数量寄存器”(386 行)。至于起始扇区，那就要看使用的是 LBA 模式（见 390 行）还是 CHS 模式（见 401 行）了。如果使用的是 CHS 模式就要作一些换算（403～404，406～407 行）。我们继续往下看：

[run_task_queue( ) > __run_task_queue( ) > generic_unplug_device( ) > __generic_unplug_device( ) > do_ide_request( ) > ide_do_request( ) > start_request( ) > do_rw_disk( )]

```
423         if (rq->cmd == READ) {
424     #ifdef CONFIG_BLK_DEV_IDEDMA
425             if (drive->using_dma && !(HWIF(drive)->dmaproc(ide_dma_read, drive)))
426                 return ide_started;
427     #endif /* CONFIG_BLK_DEV_IDEDMA */
428             ide_set_handler(drive, &read_intr, WAIT_CMD, NULL);
429             OUT_BYTE(drive->mult_count ? WIN_MULTREAD : WIN_READ, IDE_COMMAND_REG);
430             return ide_started;
431         }
```

大家知道，对外部设备的 I/O 技术有两种。一种是由 CPU 驱动的“程序控制 I/O”，其特点是：当外设已经准备好进行 I/O 时，就由 CPU 执行一段底层 I/O 驱动程序，在外设与内存缓冲区之间“搬运”数据，而外设与内存并不直接接触。另一种是由外设“直接访问内存”，即 DMA，CPU 把缓冲区的地址与需要读/写的长度告诉外设，外设在准备好以后就通过有关硬件向 CPU 发出一个 DMA 请求，要求 CPU 暂停使用内存，获得同意之后就直接在内存与外设之间传输数据，完成以后再把对内存的访问权归还给 CPU。但是，CPU 对曾经发生过的暂停使用内存以及谁在暂停期间访问了内存这些事情并无知觉，因为那是由硬件实现而不是在程序控制下实现的。所以，需要有一种手段，让 CPU 知道由某种设备驱动的一次 DMA 已经完成。那么，在程序控制 I/O 中怎样让 CPU 知道外设已经准备好，在 DMA 中怎样让 CPU 知道 DMA 已经完成呢？这又有两种方法，一种是由 CPU 查询（poll），另一种是由设备向 CPU 发出中断请求。这样，一共就有 4 种可能的组合。由 CPU 查询的方法显然是效率很低的，所以只在很简单、要求很低的系统中才使用，像 Linux 这样的系统当然要采用中断方法（见第 3 章）。于是就只剩下中断与 DMA 和中断与程序控制 I/O 两种了。是否采用 DMA 是一个系统配置的选项。如果选择了采用 DMA，这里的条件编译控制 CONFIG_BLK_DEV_IDEDMA 就有定义，否则就无定义，而在编译时跳过这里的 425 和 426 两行。我们将在以后专门讨论 DMA，在这里假定采用程序控制 I/O 的方式。

对于读操作，CPU 需要在磁盘已经准备好供读出时得到通知，再执行从磁盘读出，所以要预先设置好一个中断服务程序 read_intr( )，这是由 ide_set_handler( )完成的，它的代码在 ide.c 中：

[run_task_queue( ) > __run_task_queue( ) > generic_unplug_device( ) > __generic_unplug_device( ) > do_ide_request( ) > ide_do_request( ) > start_request( ) > do_rw_disk( ) > ide_set_handler( )]

```
525     /*
```

```
526      * This should get invoked any time we exit the driver to
527      * wait for an interrupt response from a drive.  handler( ) points
528      * at the appropriate code to handle the next interrupt, and a
529      * timer is started to prevent us from waiting forever in case
530      * something goes wrong (see the ide_timer_expiry( ) handler later on).
531      */
532     void ide_set_handler (ide_drive_t *drive, ide_handler_t *handler,
533                           unsigned int timeout, ide_expiry_t *expiry)
534     {
535         unsigned long flags;
536         ide_hwgroup_t *hwgroup = HWGROUP(drive);
537
538         spin_lock_irqsave(&io_request_lock, flags);
539         if (hwgroup->handler != NULL) {
540             printk("%s: ide_set_handler: handler not null; old=%p, new=%p\n",
541                 drive->name, hwgroup->handler, handler);
542         }
543         hwgroup->handler       = handler;
544         hwgroup->expiry        = expiry;
545         hwgroup->timer.expires  = jiffies + timeout;
546         add_timer(&hwgroup->timer);
547         spin_unlock_irqrestore(&io_request_lock, flags);
548     }
```

这里的第一和第二个参数是不言自明的；第三个参数是为本次操作所设置的时间限制；第四个参数是个函数指针，这是当超过时间限制时要调用的函数，指针为 NULL 表示采用标准的超时处理。这些参数都记录在代表着具体硬盘所属磁盘组的 ide_hwgroup_t 数据结构中。从这个函数也可以看出，一个磁盘组在同一时间里只能有一个磁盘处于操作状态，因为在 ide_hwgroup_t 数据结构中只有一个函数指针指向中断服务程序，也只有一个用于操作超时的定时器。

为中断作好准备以后，CPU 这一边就万事俱备，可以通过命令寄存器下达启动读操作的命令了（见 429 行）。由于具体磁盘的内部缓冲区大小各不相同，有的硬盘每读出一个扇区就向 CPU 发出一个中断请求，有些则可以在积累起多个扇区的内容（如果需要的话）以后才向 CPU 发出一个中断请求。对后一种硬盘其 ide_drive_t 结构中的 mult_count 字段为非 0，而硬盘除接受读单个扇区的命令 WIN_READ 外也可接受读多个扇区的命令 WIN_MULTREAD。那么，所谓“多个扇区”，到底是几个呢？这就取决于需要读的扇区总数（见 386 行）和磁盘内部缓冲区的大小。这个缓冲区大小在 IDE 设备初始化时设置在 mult_count 字段中。读命令一经发出，对读操作的启动就完成了(430 行)，所以返回一个常数 ide_started。从指定的扇区将数据读入其内部缓冲器后，磁盘就会向 CPU 发出中断请求，以后就是中断服务程序 read_intr( )的事了。

再看写操作的启动：

[run_task_queue( ) > __run_task_queue( ) > generic_unplug_device( ) > __generic_unplug_device( ) > do_ide_request( ) > ide_do_request( ) > start_request( ) > do_rw_disk( )]

```
432          if (rq->cmd == WRITE) {
```

```
433             ide_startstop_t startstop;
434     #ifdef CONFIG_BLK_DEV_IDEDMA
435             if (drive->using_dma && !(HWIF(drive)->dmaproc(ide_dma_write, drive)))
436                 return ide_started;
437     #endif /* CONFIG_BLK_DEV_IDEDMA */
438             OUT_BYTE(drive->mult_count ? WIN_MULTWRITE : WIN_WRITE, IDE_COMMAND_REG);
439             if (ide_wait_stat(&startstop, drive, DATA_READY,
                                                drive->bad_wstat, WAIT_DRQ)) {
440                 printk(KERN_ERR "%s: no DRQ after issuing %s\n", drive->name,
441                     drive->mult_count ? "MULTWRITE" : "WRITE");
442                 return startstop;
443             }
444             if (!drive->unmask)
445                 __cli( );   /* local CPU only */
446             if (drive->mult_count) {
447                 ide_hwgroup_t *hwgroup = HWGROUP(drive);
448                 /*
449                  * Ugh.. this part looks ugly because we MUST set up
450                  * the interrupt handler before outputting the first block
451                  * of data to be written.  If we hit an error (corrupted buffer list)
452                  * in ide_multwrite( ), then we need to remove the handler/timer
453                  * before returning. Fortunately, this NEVER happens (right?).
454                  *
455                  * Except when you get an error it seems...
456                  */
457                 hwgroup->wrq = *rq; /* scratchpad */
458                 ide_set_handler (drive, &multwrite_intr, WAIT_CMD, NULL);
459                 if (ide_multwrite(drive, drive->mult_count)) {
460                     unsigned long flags;
461                     spin_lock_irqsave(&io_request_lock, flags);
462                     hwgroup->handler = NULL;
463                     del_timer(&hwgroup->timer);
464                     spin_unlock_irqrestore(&io_request_lock, flags);
465                     return ide_stopped;
466                 }
467             } else {
468                 ide_set_handler (drive, &write_intr, WAIT_CMD, NULL);
469                 idedisk_output_data(drive, rq->buffer, SECTOR_WORDS);
470             }
471             return ide_started;
472         }
473         printk(KERN_ERR "%s: bad command: %d\n", drive->name, rq->cmd);
474         ide_end_request(0, HWGROUP(drive));
475         return ide_stopped;
476     }
```

同样，我们在这里先不关心 DMA。对写操作也要先发出启动命令。同样，根据具体磁盘的特性，

有针对单个扇区的 WIN_WRITE 和多个扇区的 WIN_NULTWRITE 两种命令。发出命令以后，就要通过 ide_wait_stat( )等待和读取磁盘的状态寄存器。这个等待只不过是大约 400 微毫秒的时间，所以不值得采用中断方式（磁盘也并不为此提供中断请求的功能）。另一方面，由于对 I/O 的启动是在 bh 函数中进行的，所以也不能进入睡眠。磁盘在接收到写操作命令并准备好接受待写入的数据时，就将其状态寄存器中的 DRQ 位设成 1，表示可以传输数据了。CPU 一方检测到这个信息以后，就可以开始输出数据了。可想而知，单扇区的写出与多扇区的写出会略有不同，我们把多扇区的写出(447～466 行)留给读者。这里只看与单扇区有关的代码。

首先还是设置中断服务程序，不过这一次设置的是 write_intr( )。然后就是数据的输出了，函数 idedisk_output_data( )的代码也在 ide_disk.c 中：

[run_task_queue( ) > __run_task_queue( ) > generic_unplug_device( ) > __generic_unplug_device( )
 > do_ide_request( ) > ide_do_request( ) > start_request( ) > do_rw_disk( ) > idedisk_output_data( )]

```
79    static inline void idedisk_output_data (ide_drive_t *drive, void *buffer, unsigned
      int wcount)
80    {
81        if (drive->bswap) {
82            idedisk_bswap_data(buffer, wcount);
83            ide_output_data(drive, buffer, wcount);
84            idedisk_bswap_data(buffer, wcount);
85        } else
86            ide_output_data(drive, buffer, wcount);
87    }
```

先看参数，第一个参数指向代表具体硬盘的 ide_drive_t 结构；第二个参数指向缓冲区，操作请求结构中的指针 buffer 总是指向其缓冲区队列中第一个缓冲区的开头；第三个参数是待写数据的长度，这里固定为 SECTOR_WORDS，即一个扇区的长度（以 16 位字计算）。以前讲过，同一操作请求所涉及的扇区一定是连续的，但是他们的缓冲区却不一定连续，不过缓冲区的长度一定是扇区大小的整数倍。例如，将两个操作请求合并成一个时，它们的扇区一定是连续的，但是缓冲区却不连续。

这个函数所做的还不是真正的数据输出，那是由 ide_output_data( )完成的。这里所做的是可能需要的对字节次序的转换。CPU 有 big_ending 和 little_enging 之分，16 位和 32 位的数据在这两种制式中字节次序不同。可是，磁盘上的数据制式不应该跟着 CPU 跑，而应该固定使用一种，所以有可能需要在读/写磁盘时加以转换。

函数 ide_output_data( )的代码在 ide.c 中：

[run_task_queue( ) > __run_task_queue( ) > generic_unplug_device( ) > __generic_unplug_device( )
 > do_ide_request( ) > ide_do_request( ) > start_request( ) > do_rw_disk( ) > idedisk_output_data( )
 > ide_output_data( )]

```
405   /*
406    * This is used for most PIO data transfers *to* the IDE interface
407    */
408   void ide_output_data (ide_drive_t *drive, void *buffer, unsigned int wcount)
409   {
```

```
410         byte io_32bit = drive->io_32bit;
411 
412         if (io_32bit) {
413     #if SUPPORT_VLB_SYNC
414             if (io_32bit & 2) {
415                 unsigned long flags;
416                   save_flags(flags);    /* local CPU only */
417                   cli( );       /* local CPU only */
418                 do_vlb_sync(IDE_NSECTOR_REG);
419                 outsl(IDE_DATA_REG, buffer, wcount);
420                 __restore_flags(flags); /* local CPU only */
421             } else
422     #endif /* SUPPORT_VLB_SYNC */
423                 outsl(IDE_DATA_REG, buffer, wcount);
424         } else {
425     #if SUPPORT_SLOW_DATA_PORTS
426             if (drive->slow) {
427                 unsigned short *ptr = (unsigned short *) buffer;
428                 while (wcount--) {
429                     outw_p(*ptr++, IDE_DATA_REG);
430                     outw_p(*ptr++, IDE_DATA_REG);
431                 }
432             } else
433     #endif /* SUPPORT_SLOW_DATA_PORTS */
434                 outsw(IDE_DATA_REG, buffer, wcount<<1);
435         }
436     }
```

大部分 IDE 磁盘的数据寄存器都是 16 位的，有些老式的接口还不允许操作太快而不能使用“成串输出”指令 OUTS，但也有些新式 IDE 磁盘提供 32 位数据寄存器。对大多数目前在使用中的 IDE 磁盘而言，具体的输出是通过成串输出指令（见 434 行）完成的。输出的数据写入到磁盘的内部缓冲区中。

输出完成以后，对写操作的启动就完成了。磁盘在完成本次写入操作（将数据写入指定的扇区中）后会向 CPU 发出中断请求，以后就是中断服务程序 write_intr( )的事情了。同样，这里也返回 ide_started（见上面的 471 行）。

回到前面 start_request( )的代码中（见 1189 行）。对于有特殊要求的磁盘，其 special_all 字段可能为非 0，所以会走另一条路线执行 do_special( )。不过，这些特殊的处理都带有初始化的性质，因而执行了一遍以后就把相应的标志位清成 0，以后 special.all 就为 0 而走正常的路线了。函数 do_special( ) 的代码也在 ide.c 中，我们把它留给有兴趣的读者自己阅读。

至此，对块设备 I/O 的启动已经完成，以后的事就完全是异步的了。下面我们以读操作为例来考察其全过程，读者在理解了该操作的全过程以后自然不难把它推广到写操作。

磁盘从指定的扇区中将数据读入其内部缓冲区以后，就向 CPU 发出中断请求。在 IDE 设备初始化时，从 hwif_init( )调用的 init_irq( )中，已经通过 request_irq( )为每个 IDE 磁盘组登记了总的中断服务程序 ide_intr( )，所以 CPU 在响应中断时就经由内核的中断服务机制（见第 3 章）进入了 ide_intr( )，它

的代码在 ide.c 中：

[do_IRQ( ) > handle_IRQ_event( ) > ide_intr( )]

```
1524    /*
1525     * entry point for all interrupts, caller does __cli( ) for us
1526     */
1527    void ide_intr (int irq, void *dev_id, struct pt_regs *regs)
1528    {
1529        unsigned long flags;
1530        ide_hwgroup_t *hwgroup = (ide_hwgroup_t *)dev_id;
1531        ide_hwif_t *hwif;
1532        ide_drive_t *drive;
1533        ide_handler_t *handler;
1534        ide_startstop_t startstop;
1535
1536        spin_lock_irqsave(&io_request_lock, flags);
1537        hwif = hwgroup->hwif;
1538
1539        if (!ide_ack_intr(hwif)) {
1540            spin_unlock_irqrestore(&io_request_lock, flags);
1541            return;
1542        }
1543
1544        if ((handler = hwgroup->handler) == NULL || hwgroup->poll_timeout != 0) {
1545            /*
1546             * Not expecting an interrupt from this drive.
1547             * That means this could be:
1548             *  (1) an interrupt from another PCI device
1549             *  sharing the same PCI INT# as us.
1550             * or   (2) a drive just entered sleep or standby mode,
1551             *  and is interrupting to let us know.
1552             * or   (3) a spurious interrupt of unknown origin.
1553             *
1554             * For PCI, we cannot tell the difference,
1555             * so in that case we just ignore it and hope it goes away.
1556             */
1557    #ifdef CONFIG_BLK_DEV_IDEPCI
1558            if (IDE_PCI_DEVID_EQ(hwif->pci_devid, IDE_PCI_DEVID_NULL))
1559    #endif  /* CONFIG_BLK_DEV_IDEPCI */
1560            {
1561                /*
1562                 * Probably not a shared PCI interrupt,
1563                 * so we can safely try to do something about it:
1564                 */
1565                unexpected_intr(irq, hwgroup);
1566    #ifdef CONFIG_BLK_DEV_IDEPCI
1567            } else {
```

```
1568            /*
1569            * Whack the status register, just in case we have a leftover pending IRQ.
1570            */
1571                (void) IN_BYTE(hwif->io_ports[IDE_STATUS_OFFSET]);
1572    #endif /* CONFIG_BLK_DEV_IDEPCI */
1573            }
1574            spin_unlock_irqrestore(&io_request_lock, flags);
1575            return;
1576        }
1577        drive = hwgroup->drive;
1578        if (!drive) {
1579    /*
1580    * This should NEVER happen, and there isn't much we could do about it here.
1581    */
1582            spin_unlock_irqrestore(&io_request_lock, flags);
1583            return;
1584        }
1585        if (!drive_is_ready(drive)) {
1586        /*
1587        * This happens regularly when we share a PCI IRQ with another device.
1588        * Unfortunately, it can also happen with some buggy drives that trigger
1589        * the IRQ before their status register is up to date.  Hopefully we have
1590        * enough advance overhead that the latter isn't a problem.
1591        */
1592            spin_unlock_irqrestore(&io_request_lock, flags);
1593            return;
1594        }
1595        if (!hwgroup->busy) {
1596            hwgroup->busy = 1;  /* paranoia */
1597            printk("%s: ide_intr: hwgroup->busy was 0 ??\n", drive->name);
1598        }
1599        hwgroup->handler = NULL;
1600        del_timer(&hwgroup->timer);
1601        spin_unlock(&io_request_lock);
1602
1603        if (drive->unmask)
1604            ide__sti( );    /* local CPU only */
1605        startstop = handler(drive);     /* service this interrupt, may set
                                               handler for next interrupt */
1606        spin_lock_irq(&io_request_lock);
1607
1608        /*
1609         * Note that handler( ) may have set things up for another
1610         * interrupt to occur soon, but it cannot happen until
1611         * we exit from this routine, because it will be the
1612         * same irq as is currently being serviced here, and Linux
1613         * won't allow another of the same (on any CPU) until we return.
1614         */
```

```
1615        set_recovery_timer(HWIF(drive));
1616        drive->service_time = jiffies - drive->service_start;
1617        if (startstop == ide_stopped) {
1618            if (hwgroup->handler == NULL) { /* paranoia */
1619                hwgroup->busy = 0;
1620                ide_do_request(hwgroup, hwif->irq);
1621            } else {
1622                printk("%s: ide_intr: huh? expected NULL handler on exit\n",
                    drive->name);
1623            }
1624        }
1625        spin_unlock_irqrestore(&io_request_lock, flags);
1626    }
```

向内核的中断机制登记中断服务程序时，可以同时为之设置一个 void 指针，作为将来调用该服务程序时的参数（见第 3 章）。这就是这里的指针 dev_id，它实际上是指向磁盘组的 ide_hwgroup_t 数据结构，从这个数据结构中可以找到该磁盘组的当前 IDE 接口（见 1537 行）。对于 i386 结构的 CPU，这里的 ide_ack_intr( )是个空函数，见 ide.h：

```
109    #define ide_ack_intr(hwif)      (1)
```

由于我们已经在前面的 do_rw_disk( )中设置好了具体的中断服务程序，对于读操作是 read_intr( )，这里的函数指针 hwgroup->handler 不可能为 0，所以我们跳过第 1544 行条件语句的执行部分。同时，1578 行的指针 hwgroup->drive 也不会是 0，因为磁盘之所以会发出中断请求是因为曾经向它发出了命令。但是，即使对于在正常情况下不会发生的事也要作好准备，以防万一。同样的道理也适用于 1585 行对磁盘状态的测试。

函数指针 hwgroup->handler 的内容已经转移到另一个指针 handler 中，1599 行将其清 0。这说明在磁盘组层次上中断服务程序的设置是一次性的，以后的中断服务程序是什么要视本次中断服务中的处理而定。向磁盘发出命令启动其写操作时，曾经为磁盘的操作设置下一个定时器，以备在操作超过时间限制时采取必要的措施。现在中断既已发生（并且已经检查了状态寄存器），这个定时器显然已经不需要了，所以通过 del_timer( )将其撤销（1600 行）。

接着就是调用预先设置好的中断服务程序了（1605 行），对于读盘操作这是 read_intr( )，其代码在 ide_disk.c 中：

[do_IRQ( ) > handle_IRQ_event( ) > ide_intr( ) > read_intr( )]

```
134    /*
135     * read_intr( ) is the handler for disk read/multread interrupts
136     */
137    static ide_startstop_t read_intr (ide_drive_t *drive)
138    {
139        byte stat;
140        int i;
141        unsigned int msect, nsect;
```

```
142         struct request *rq;
143 
144         /* new way for dealing with premature shared PCI interrupts */
145         if (!OK_STAT(stat=GET_STAT( ),DATA_READY,BAD_R_STAT)) {
146             if (stat & (ERR_STAT|DRQ_STAT)) {
147                 return ide_error(drive, "read_intr", stat);
148             }
149             /* no data yet, so wait for another interrupt */
150             ide_set_handler(drive, &read_intr, WAIT_CMD, NULL);
151             return ide_started;
152         }
153         msect = drive->mult_count;
154 
155     read_next:
156         rq = HWGROUP(drive)->rq;
157         if (msect) {
158             if ((nsect = rq->current_nr_sectors) > msect)
159                 nsect = msect;
160             msect -= nsect;
161         } else
162             nsect = 1;
163         idedisk_input_data(drive, rq->buffer, nsect * SECTOR_WORDS);
164     #ifdef DEBUG
165         printk("%s:  read: sectors(%ld-%ld), buffer=0x%08lx, remaining=%ld\n",
166             drive->name, rq->sector, rq->sector+nsect-1,
167             (unsigned long) rq->buffer+(nsect<<9), rq->nr_sectors-nsect);
168     #endif
169         rq->sector += nsect;
170         rq->buffer += nsect<<9;
171         rq->errors = 0;
172         i = (rq->nr_sectors -= nsect);
173         if (((long)(rq->current_nr_sectors -= nsect)) <= 0)
174             ide_end_request(1, HWGROUP(drive));
175         if (i > 0) {
176             if (msect)
177                 goto read_next;
178             ide_set_handler (drive, &read_intr, WAIT_CMD, NULL);
179                     return ide_started;
180         }
181             return ide_stopped;
182     }
```

这里的 OK_STAT( )是个宏操作，它将由 GET_STAT( )从状态寄存器读回的数值与 DATA_READY 和 BAD_R_STAT 比较，看是否状态字节中的 DATA_READY 位为 1 而 BAD_R_STAT 位为 0。如果状态字节表明磁盘尚未准备好，但是也没有出错（见 150 行），就再把中断服务程序设置成 read_intr( )，到下次中断时再来处理。

如果磁盘确已准备好了，那就从相应的 ide_hwgroup_t 结构中找到当前的操作请求。以前讲过：每

个 IDE 设备都有自己的操作请求队列，但是整个 IDE 接口组只能有一个“当前操作请求”，ide_hwgroup_t 结构中的指针 rq 就指向这个请求的 request 数据结构。所以，这里先要通过 HWGROUP( )找到设备所属的接口组，再取得其当前操作请求。此外，还要根据反映磁盘内部缓冲区大小的 drive->mult_count 和本次操作中要求从磁盘读入当前缓冲区的扇区个数 rq->current_nr_sectors，确定一次从磁盘读入的扇区个数为 nsect。

函数 idedisk_input_data( )的代码在 ide_disk.c 中。可以想像，它与前面的 idedisk_output_data( )相似而只是传输数据的方向相反，所以我们不看具体的代码了。

从磁盘读入以后，要对当前操作请求的指针 buffer 和计数器 sector、nr_sectors、以及 current_nr_sectors 作相应的调整(见 169～170 以及 172～173 行)，并计算出当前操作尚未完成的扇区数(172 行)。这里 sector 为已经读入的累计扇区数(对本次操作请求)；nr_sectors 为本次操作请求要读入的扇区总数；current_nr_sectors 为本次中断服务中要从磁盘读入到当前内存缓冲区的扇区数；而 msect 则为本次操作中要求磁盘读入，但尚未读入到内存缓冲区的扇区数。即使磁盘支持多扇区操作，一个操作请求也可能要通过好几次中断服务才能完成。此外，一个操作请求也可能涉及多个缓冲区，所以在 request 数据结构中有个缓冲区队列；对分布在不同缓冲区中的连续扇区可以合并成同一次多扇区操作，但是却要分次从磁盘读入，特别是考虑到采用 DMA 时更是如此，因为一般的 DMA 控制器(不是所谓 Intelegent DMA)没有自动切换缓冲区的功能。

考虑已经读进来的这部分数据与整体的关系，存在着三种可能的情况：

(1) 这些数据只是一部分，并且磁盘的内部缓冲区中还有数据尚未读出，之所以不能把内部缓冲区中的数据全部读出是因为已经到了一个内存缓冲段的末尾。在继续从磁盘的内部缓冲区读出之前，先要调整缓冲区指针使其指向下一个内存缓冲区，这是由 ide_end_request( )完成的（见 173～174 行和 177 行）。

(2) 由于磁盘内部缓冲区大小的限制，磁盘内部缓冲区中的数据已经全部读出，但是当前操作尚未完成，磁盘会继续从有关扇区将数据读入内部缓冲区后再次发出中断请求，所以要为下一次中断作好准备（见 178～179 行）。同时，如果恰巧已经到了当前缓冲区的末尾，就也要通过 ide_end_request( )启用新的缓冲区（见 173～174 行）。

(3) 这些数据就是当前操作请求所要求的全部，或者是它的最后一部分，所以整个操作请求已经完成（见 181 行）。

显然，当整个操作请求完成时，内存中的缓冲区指针也必定恰好到达一个缓冲区的末尾，所以也会调用 ide_end_request( )(见 173 和 174 行)。

可见，只要到达了一个缓冲区的末尾，就会调用 ide_end_request( )，其目的可能只是启用同一操作请求中的下一个缓冲区，也可能对整个操作请求的善后处理。它的代码在 ide.c 中：

[do_IRQ( ) > handle_IRQ_event( ) > ide_intr( ) > read_intr( ) > ide_end_request( )]

```
505    /*
506     * This is our end_request replacement function.
507     */
508    void ide_end_request (byte uptodate, ide_hwgroup_t *hwgroup)
509    {
510        struct request *rq;
```

```
511         unsigned long flags;
512
513         spin_lock_irqsave(&io_request_lock, flags);
514         rq = hwgroup->rq;
515
516         if (!end_that_request_first(rq, uptodate, hwgroup->drive->name)) {
517             add_blkdev_randomness(MAJOR(rq->rq_dev));
518             blkdev_dequeue request(rq);
519                 hwgroup->rq = NULL;
520             end_that_request last(rq);
521         }
522         spin_unlock_irqrestore(&io_request_lock, flags);
523     }
```

首先调用 end_that_request_first( )，其代码在 ll_rw_blk.c 中：

[do_IRQ( ) > handle_IRQ_event( ) > ide_intr( ) > read_intr( ) > ide_end_request( ) > end_that_request_first( )]

```
1099    /*
1100     * First step of what used to be end_request
1101     *
1102     * 0 means continue with end_that_request_last,
1103     * 1 means we are done
1104     */
1105
1106    int end_that_request_first (struct request *req, int uptodate, char *name)
1107    {
1108        struct buffer_head * bh;
1109        int nsect;
1110
1111        req->errors = 0;
1112        if (!uptodate)
1113            printk("end_request: I/O error, dev %s (%s), sector %lu\n",
1114                kdevname(req->rq_dev), name, req->sector);
1115
1116        if ((bh = req->bh) != NULL) {
1117            nsect = bh->b_size >> 9;
1118            req->bh = bh->b_reqnext;
1119            bh->b_reqnext = NULL;
1120            bh->b_end_io(bh, uptodate);
1121            if ((bh = req->bh) != NULL) {
1122                req->hard_sector += nsect;
1123                req->hard_nr sectors -= nsect;
1124                req->sector = req->hard_sector;
1125                req->nr_sectors = req->hard_nr_sectors;
1126
1127                req->current_nr_sectors = bh->b_size >> 9;
```

```
1128                if (req->nr_sectors < req->current_nr_sectors) {
1129                    req->nr_sectors = req->current_nr_sectors;
1130                    printk("end_request: buffer-list destroyed\n");
1131                }
1132                req->buffer = bh->b_data;
1133                return 1;
1134            }
1135        }
1136        return 0;
1137    }
```

不管整个操作请求是否已经完成，对其中一个缓冲区的读入总归是完成了，否则就到不了这个函数中，所以把当前操作请求的缓冲区指针移向队列中的下一个缓冲区(1118 行)，将原来的当前缓冲区从队列中解脱出来。同时，既然对一个缓冲区的操作已经完成，就要通过其函数指针 b_end_io 调用该缓冲区的善后程序。对于写操作，这个函数指针是在__block_prepare_write( )以及类似的函数中设置的；对于读操作则在 getblk( )一类的函数中通过 init_buffer( )设置，但是这个函数指针一般都指向 end_buffer_io_sync( )，它的代码在 fs/buffer.c 中：

[do_IRQ( ) > handle_IRQ_event( ) > ide_intr( ) > read_intr( ) > ide_end_request( )
 > end_that_request_first( ) > end_buffer_io_sync( )]

```
978    /*
979     * Default IO end handler, used by "ll_rw_block( )".
980     */
981    static void end_buffer_io_sync(struct buffer_head *bh, int uptodate)
982    {
983        mark_buffer_uptodate(bh, uptodate);
984        unlock_buffer(bh);
985    }
```

第一件事是将已经读/写完毕的缓冲区的 BH_Uptodate 标志位设成 1，表示该缓冲区的内容已是最新版本了。第二件事是 unlock_buffer( )，见 locks.h：

[do_IRQ( ) > handle_IRQ_event( ) > ide_intr( ) > read_intr( ) > ide_end_request( )
 > end_that_request_first( ) > end_buffer_io_sync( ) > unlock_buffer( )]

```
29    extern inline void unlock_buffer(struct buffer_head *bh)
30    {
31        clear_bit(BH_Lock, &bh->b_state);
32        smp_mb__after_clear_bit( );
33        if (waitqueue_active(&bh->b_wait))
34            wake_up(&bh->b_wait);
35    }
```

它一方面清除缓冲区的 BH_lock 标志位，一方面唤醒被锁在外面、正在睡眠等待的进程。哪些进程会被唤醒呢？至少启动了对该缓冲区的读/写操作，并调用了 wait_on_buffer( )正在等待其完成的进程

会被唤醒。例如，在 block_write( )和 block_read( )的代码中，当前进程在通过 ll_rw_block( )发出对块设备的读/写请求以后都要调用 wait_on_buffer( )等待。所以，wait_on_buffer( )是启动操作的进程与异步的设备操作过程的同步点。当读/写操作涉及多个缓冲区时，对这个函数的调用通常放在一个循环中，以取得与所有缓冲区（读/写）的同步。

回到 end_that_request_first( )的代码中。如果缓冲区队列中还有下一个缓冲区存在，就相应地设置 request 数据结构中的各项参数，为下一个缓冲区的读/写作好准备，并返回 1。如果缓冲区队列已经空了就返回 0，说明对整个操作请求的服务都已完成。

回到 ide_end_request( )的代码中。如果 end_that_request_first( )返回的是 1，就表示同一操作请求的缓冲区队列中还有下一个缓冲区，并且已经启用。既然操作尚未完成，回到 read_intr( )中自会进一步加以处理。反之，如果返回的是 0，那就说明整个操作请求都已完成了。这时候需要执行一些对整个操作请求的善后处理。处理些什么呢？一是通过 blkdev_dequeue_request( )将代表这个操作请求的 request 结构从操作请求队列中摘除，并将整个 IDE 接口组的当前请求指针设置成 NULL。二是调用 end_that_request_last( )，等一下我们要看它的源代码。除此之外，还搭配了一点“私货”，就是对 add_blkdev_randomness( )的调用。这是干什么用的呢？系统中常常需要生成一些随机数，为了使这些“随机数”更加随机，需要在生成的过程中介入尽可能多、尽可能随机的因素，称为“熵”。例如，键盘输入的内容就可以用来作为这些因素之一，因为每次开机以后键盘输入的内容和次序往往是不完全一样的。而对块设备的访问也可以用来作为一个因素，这就是这里调用这个函数的原因。

函数 end_that_request_last( )的代码在 ll_rw_blk.c 中：

[do_IRQ( ) > handle_IRQ_event( ) > ide_intr( ) > read_intr( ) > ide_end_request( ) > end_that_request_last( )]

```
1139    void end_that_request_last(struct request *req)
1140    {
1141        if (req->e) {
1142            printk("end_that_request_last called with non-dequeued req\n");
1143            BUG( );
1144        }
1145        if (req->sem != NULL)
1146            up(req->sem);
1147
1148        blkdev_release_request(req);
1149    }
```

显然，这里主要的操作是释放 request 结构（1148 行）和对内核信号量 req->sem 的 up( )操作。用户进程可能通过系统调用 ioctl( )直接启动一些对 IDE 设备的操作，对这些操作要通过使用内核信号量来保证互斥，现在既然操作已经完成就要通过 up( )退出临界区。

回到 read_intr( )中(175 行)，如果还有扇区尚未读入，那就是因为原来的缓冲区已经满了，现在已经通过 ide_end_request( )在 end_that_request_first( )中切换到了下一个缓冲区，所以可以继续了。此时若 msect 非 0，就表示磁盘的内部缓冲区中还有数据尚未读出，所以直接转到标号 read_next 处(155 行)继续读出。否则，就要调用 ide_set_handler( )再次设置中断服务程序，为下一次中断作好准备，这个函数的代码已经在前面看到过了。

当 CPU 从 read_intr( )返回到 ide_intr( )中时(见 ide_intr( )中的 1606 行)，其返回值有两种可能。一种是 ide_started，表示对当前操作请求的服务尚未完成，所以已经为下一次中断请求设置好了中断服务请求。不过，只要还没有从 ide_intr( )返回，就不会因新的中断请求而嵌套进入这个服务程序。另一种是 ide_stopped，表示当前操作已经完成，代表当前操作请求的 request 数据结构也已经从操作请求队列中摘除。不管返回的值是什么，有两件事总是要做的。第一件事是通过 set_recovery_timer( )根据当前的时间设置相应 ide_hwif_t 数据结构中的 last_time 字段。不过这只有对那些在两次操作之间需要有一个恢复时间的磁盘才必须。第二件事是根据当前时间和本次操作的启动时间计算出本次操作已经耗费的时间 service_time。

如果从 read_intr( )返回的值是 ide_stopped，那就要调用 ide_do_request( )，看看同一 IDE 接口组中是否还有哪个磁盘的队列中还有操作请求。这个函数的代码已经在前面读过了。

就这样，一直要到整个 IDE 接口组中再没有任何活跃（不在睡眠中）的磁盘还有操作请求时为止，到那时由中断驱动的设备读/写过程才会结束。如果结束时有磁盘正在睡眠中，而该磁盘的操作请求队列中又有操作请求，那么在睡眠到点时又会启动这个过程。当然，新操作请求的到来也会再次启动这个过程。

在前面的叙述和讨论中，我们假定采用的是程序控制 I/O 而不是 DMA，现在我们再来看看对 IDE 硬盘的 DMA 输入/输出是怎样实现的。如前所述，是否采用 DMA 是一个系统配置的选项，由条件编译控制量 CONFIG_BLK_DEV_IDEDMA 决定实际采用的代码。

大家知道，DMA 是“直接访问内存”的缩写，表示由外部设备直接访问内存。但是，在传统的 PC 系统结构中，DMA 操作要通过一个“DMA 控制器”(更确切地说是 DMA 控制器中的一个“通道”)才能进行。在这种系统结构中，要对外设进行 DMA 操作时，DMA 控制器(经过 CPU 的设置和启动)就暂时“接管”CPU 的角色，成为内存的“主设备”，代替 CPU 发出访问内存和外设寄存器所需的地址以及控制脉冲，只是操作的速度比 CPU 更快。严格地说，在这种系统结构中，外设并没有变成内存的“主设备”而“当家作主”，从而“直接”访问内存，而是仍然处于被动的地位，只不过“主设备”从 CPU 变成了 DMA 控制器而已。所以，从外部设备的角度而言，比之别的一些系统结构，传统 PC 系统结构中的 DMA 有些名不副实。然而，PCI 总线的设计推广了 PC 系统结构中 DMA 操作的概念。以前我们曾提到，PCI 总线允许连接的总线上的设备竞争成为总线主设备，并且可以把内存看成是总线上的从设备。这样一来，PCI 设备就有可能真正当家作主，作为主设备来直接访问内存了。当然，有这种能力的 PCI 设备接口的结构要更复杂一些。因具体设备性质的不同，PCI 设备(接口)可以做成具有成为总线主设备的能力，也可以不具备这种能力。对于可以成为总线主设备的设备，在配置寄存器组的命令寄存器中有个控制位，可以允许或不允许该设备(在需要时)参加竞争。IDE 硬盘(接口)就具有成为总线主设备的能力，因而可以进行真正意义上的 DMA 操作。为了与传统 PC 系统结构的 DMA 操作相区别，这种 DMA 操作称为“总线主 DMA”(Bus Master DMA)或 BMDMA。至于不具备 BMDMA 功能的 PCI 设备，如果有必要的话，仍可以通过 DMA 控制器对其进行“DMA”操作。

为了 BMDMA 的需要，IDE 接口中增设了两组 DMA 寄存器，分别用于 IDE 接口上的两个通道。这些寄存器采用 I/O 地址，并且固定为 IDE 接口的第五个地址区间(配置寄存器组中的 resource[4])。而且，IDE 接口不但能进行传统的、单缓冲区的 DMA，还能进行多缓冲区间(一个区间中可以包含若干连续的缓冲区)的“串式”DMA。只要为之准备下一个“DMA 区间表”，表中逐个地列出用于 DMA 操

作的若干缓冲区间(包括起点与长度)，并把这个表的起始地址写入 IDE 接口上相应的“总线主 IDE 描述表指针”寄存器(BMIDTPX)，则启动 DMA 操作以后 IDE 接口会先从表中找到第一个缓冲区间的地址；完成对第一个缓冲区间的操作以后就会自动转到第二个缓冲区间；如此等等，直到完成对所有缓冲区间的操作。下面读者就会看到，一旦准备好了 DMA 区间表，实现 DMA 操作的代码就很简单了。

采用 DMA 与否只涉及很底层的代码，对具体设备驱动程序的结构并无显著的影响。对于 IDE 硬盘的驱动，区别主要在底层函数 do_rw_disk( )中，为方便阅读，我们再把这个函数的有关片断列出于下(drivers/ide/ide-disk.c)：

```
377     /*
378      * do_rw_disk( ) issues READ and WRITE commands to a disk,
379      * using LBA if supported, or CHS otherwise, to address sectors.
380      * It also takes care of issuing special DRIVE_CMDs.
381      */
382     static ide_startstop_t do_rw_disk (ide_drive_t *drive,
                                     struct request *rq, unsigned long block)
383     {
        . . . . . .
423         if (rq->cmd == READ) {
424     #ifdef CONFIG_BLK_DEV_IDEDMA
425             if (drive->using_dma && !(HWIF(drive)->dmaproc(ide_dma_read, drive)))
426                 return ide_started;
427     #endif /* CONFIG_BLK_DEV_IDEDMA */
428             ide_set_handler(drive, &read_intr, WAIT_CMD, NULL);
429             OUT_BYTE(drive->mult_count ? WIN_MULTREAD : WIN_READ, IDE_COMMAND_REG);
430             return ide_started;
431         }
432         if (rq->cmd == WRITE) {
433             ide_startstop_t startstop;
434     #ifdef CONFIG_BLK_DEV_IDEDMA
435             if (drive->using_dma && !(HWIF(drive)->dmaproc(ide_dma_write, drive)))
436                 return ide_started;
437     #endif /* CONFIG_BLK_DEV_IDEDMA */
        . . . . . .
476     }
```

在 423 行以前的代码已经对 IDE 硬盘的有关寄存器(如要读/写的块号等)进行了必要的设置。有关详情可参看前面对程序控制 I/O 方式的说明。现在，剩下的只是对读/写操作的启动了。正是在这一步上，DMA 与程序控制 I/O 有了区别。

当采用 DMA 时，对 IDE 硬盘的读/写操作都是通过 ide_hwif_t 数据结构中提供的函数指针 dmaproc 完成，只是参数 func 一为 ide_dma_read，一为 ide_dma_write。这个函数指针在初始化时设置成指向 ide_dmaproc( )，其代码在 drivers/ide/ide-dma.c 中：

[do_rw_disk( ) > ide_dmaproc( )]

```
447     /*
448      * ide_dmaproc( ) initiates/aborts DMA read/write operations on a drive.
449      *
450      * The caller is assumed to have selected the drive and programmed the drive's
451      * sector address using CHS or LBA.  All that remains is to prepare for DMA
452      * and then issue the actual read/write DMA/PIO command to the drive.
453      *
454      * For ATAPI devices, we just prepare for DMA and return. The caller should
455      * then issue the packet command to the drive and call us again with
456      * ide_dma_begin afterwards.
457      *
458      * Returns 0 if all went well.
459      * Returns 1 if DMA read/write could not be started, in which case
460      * the caller should revert to PIO for the current request.
461      * May also be invoked from trm290.c
462      */
463     int ide_dmaproc (ide_dma_action_t func, ide_drive_t *drive)
464     {
465         ide_hwif_t *hwif = HWIF(drive);
466         unsigned long dma_base = hwif->dma_base;
467         byte unit = (drive->select.b.unit & 0x01);
468         unsigned int count, reading = 0;
469         byte dma_stat;
470
471         switch (func) {
472             case ide_dma_off:
473                 printk("%s: DMA disabled\n", drive->name);
474             case ide_dma_off_quietly:
475                 outb(inb(dma_base+2) & ~(1<<(5+unit)), dma_base+2);
476             case ide_dma_on:
477                 drive->using_dma = (func == ide_dma_on);
478                 if (drive->using_dma)
479                     outb(inb(dma_base+2)|(1<<(5+unit)), dma_base+2);
480                 return 0;
481             case ide_dma_check:
482                 return config_drive_for_dma (drive);
483             case ide_dma_read:
484                 reading = 1 << 3;
485             case ide_dma_write:
486                 SELECT_READ_WRITE(hwif,drive,func);
487                 if (!(count = ide_build_dmatable(drive, func)))
488                     return 1;    /* try PIO instead of DMA */
489                 outl(hwif->dmatable_dma, dma_base + 4); /* PRD table */
490                 outb(reading, dma_base);              /* specify r/w */
491                 outb(inb(dma_base+2)|6, dma_base+2);/* clear INTR & ERROR flags */
492                 drive->waiting_for_dma = 1;
493                 if (drive->media != ide_disk)
494                     return 0;
```

```
495                 ide_set_handler(drive, &ide_dma_intr, WAIT_CMD, dma_timer_expiry);
                        /* issue cmd to drive */
496                 OUT_BYTE(reading ? WIN_READDMA : WIN_WRITEDMA, IDE_COMMAND_REG);
497             case ide_dma_begin:
498                 /* Note that this is done *after* the cmd has
499                  * been issued to the drive, as per the BM-IDE spec.
500                  * The Promise Ultra33 doesn't work correctly when
501                  * we do this part before issuing the drive cmd.
502                  */
503                 outb(inb(dma_base)|1, dma_base);        /* start DMA */
504                 return 0;
505             case ide_dma_end: /* returns 1 on error, 0 otherwise */
506                 drive->waiting_for_dma = 0;
507                 outb(inb(dma_base)&~1, dma_base);   /* stop DMA */
508                 dma_stat = inb(dma_base+2);      /* get DMA status */
509                 outb(dma_stat|6, dma_base+2);/* clear the INTR & ERROR bits */
510                 ide_destroy_dmatable(drive);     /* purge DMA mappings */
511                 return (dma_stat & 7) != 4; /* verify good DMA status */
512             case ide_dma_test_irq: /* returns 1 if dma irq issued, 0 otherwise */
513                 dma_stat = inb(dma_base+2);
514     #if 0   /* do not set unless you know what you are doing */
515                 if (dma_stat & 4) {
516                     byte stat = GET_STAT( );
517                     outb(dma_base+2, dma_stat & 0xE4);
518                 }
519     #endif
520                 return (dma_stat & 4) == 4; /* return 1 if INTR asserted */
521             case ide_dma_bad_drive:
522             case ide_dma_good_drive:
523                 return check_drive_lists(drive, (func == ide_dma_good_drive));
524             case ide_dma_verbose:
525                 return report_drive_dmaing(drive);
526             case ide_dma_timeout:
527     #ifdef CONFIG_BLK_DEV_IDEDMA_TIMEOUT
528                 /*
529                  * Have to issue an abort and requeue the request
530                  * DMA engine got turned off by a goofy ASIC, and
531                  * we have to clean up the mess, and here is as good
532                  * as any.  Do it globally for all chipsets.
533                  */
534     #endif /* CONFIG_BLK_DEV_IDEDMA_TIMEOUT */
535             case ide_dma_retune:
536             case ide_dma_lostirq:
537                 printk("ide_dmaproc: chipset supported %s func only: %d\n",
                                                ide_dmafunc_verbose(func),  func);
538                 return 1;
539             default:
540                 printk("ide_dmaproc: unsupported %s func: %d\n",
```

```
                                            ide dmafunc verbose(func), func);
541                 return 1;
542         }
543     }
```

我们在这里只关心 ide_dma_read 和 ide_dma_write 两种操作，注意在 484 行下面没有 break 或 return 语句，读和写的区别仅在于变量 reading 的值为 0x8 或 0x0，IDE 硬盘接口的 BMDMA 命令寄存器中用这一位区分操作的方向。此外，在 496 行后面也没有 break 或 return 语句，也就是说 ide_dma_read 或 ide_dma_write 操作都蕴含着 ide_dma_begin 操作。宏操作 SELECT_READ_WRITE 定义于 include/linux/ide.h：

```
203     #define SELECT_READ_WRITE(hwif, drive, func)              \
204     {                                             \
205         if (hwif->rwproc)                           \
206             hwif->rwproc(drive, func);              \
207     }
```

其目的是为一些特殊的 IDE 硬盘提供可能需要的附加操作，一般的 IDE 硬盘不需要附加操作，这个函数指针 rwproc 就为 0。

启动 DMA 操作之前，先要通过 ide_build_dmatable( )准备好 DMA 区间表。这个函数的代码在 drivers/ide/ide-dma.c 中：

[do_rw_disk( ) > ide_dmaproc( ) > ide_build_dmatable( )]

```
248     int ide_build_dmatable (ide_drive_t *drive, ide_dma_action_t func)
249     {
250         unsigned int *table = HWIF(drive)->dmatable cpu;
251     #ifdef CONFIG_BLK_DEV_TRM290
252         unsigned int is_trm290_chipset = (HWIF(drive)->chipset == ide_trm290);
253     #else
254         const int is_trm290_chipset = 0;
255     #endif
256         unsigned int count = 0;
257         int i;
258         struct scatterlist *sg;
259
260         HWIF(drive)->sg nents = i = ide build sglist(HWIF(drive), HWGROUP(drive)->rq);
261
262         sg = HWIF(drive)->sg_table;
263         while (i && sg_dma_len(sg)) {
264             u32 cur_addr;
265             u32 cur_len;
266
267             cur_addr = sg_dma_address(sg);
268             cur_len = sg_dma_len(sg);
```

```
269
270         /*
271          * Fill in the dma table, without crossing any 64kB boundaries.
272          * Most hardware requires 16-bit alignment of all blocks,
273          * but the trm290 requires 32-bit alignment.
274          */
275
276         while (cur_len) {
277             if (++count >= PRD_ENTRIES) {
278                 printk("%s: DMA table too small\n", drive->name);
279                 pci_unmap_sg(HWIF(drive)->pci_dev,
280                              HWIF(drive)->sg_table,
281                              HWIF(drive)->sg_nents,
282                              HWIF(drive)->sg_dma_direction);
283                 return 0; /* revert to PIO for this request */
284             } else {
285                 u32 xcount, bcount = 0x10000 - (cur_addr & 0xffff);
286
287                 if (bcount > cur_len)
288                     bcount = cur_len;
289                 *table++ = cpu_to_le32(cur_addr);
290                 xcount = bcount & 0xffff;
291                 if (is_trm290_chipset)
292                     xcount = ((xcount >> 2) - 1) << 16;
293                 *table++ = cpu_to_le32(xcount);
294                 cur_addr += bcount;
295                 cur_len -= bcount;
296             }
297         }
298
299         sg++;
300         i--;
301     }
302
303     if (!count)
304         printk("%s: empty DMA table?\n", drive->name);
305     else if (!is_trm290_chipset)
306         *--table |= cpu_to_le32(0x80000000);
307
308     return count;
309 }
```

在 ide_hwif_t 数据结构中有两个指针，即 dmatable_cpu 和 dmatable_dma，二者都指向同一个用于 DMA 区间表的页面，这个页面是在初始化时分配好的。所不同的是，dmatable_cpu 通过页面的虚拟地址指向这个页面，这是 CPU 所看到的 DMA 区间表；而 dmatable_dma 则通过其物理地址指向这个页面，这是 IDE 接口的 DMA 功能部分所看到的 DMA 区间表。同时，ide_hwif_t 数据结构中还有个指针 sg_table，指向一个 scatterlist 结构数组。每个 scatterlist 结构都描述了一个用于 DMA 操作的缓冲区间，

包括其起始(虚拟)地址 address 与长度 length。如前所述，每个缓冲区间可以包含若干连续的缓冲区，而 IDE 接口可以对若干个散布的(所以称为 scatterlist) 缓冲区间操作。当文件系统层要求设备驱动层完成对具体设备的 I/O 时，交下来的是一个 buffer_head 结构队列，队列中的每个 buffer_head 数据结构都描述着一个缓冲区。这些缓冲区中有些是互相连续的，有些则不是，所以先要把它们整理成若干个缓冲区间，建立起一个 scatterlist，为进一步建立 DMA 区间表作好准备。这种数据结构定义于 include/asm-i386/scatterlist.h：

```
4    struct scatterlist {
5        char *  address;     /* Location data is to be transferred to */
6        char * alt_address; /* Location of actual if address is a
7                             * dma indirect buffer.  NULL otherwise */
8        unsigned int length;
9    };
```

先通过 ide_build_sglist( )根据具体的操作请求建立起一个“物理区间表”，把同一个操作请求中互相连续的缓冲区都合并成缓冲区间，这个函数的代码在 drivers/ide/ide-dma.c 中：

```
214    static int ide_build_sglist (ide_hwif_t *hwif, struct request *rq)
215    {
216        struct buffer_head *bh;
217        struct scatterlist *sg = hwif->sg_table;
218        int nents = 0;
219
220        if (rq->cmd == READ)
221            hwif->sg_dma_direction = PCI_DMA_FROMDEVICE;
222        else
223            hwif->sg_dma_direction = PCI_DMA_TODEVICE;
224        bh = rq->bh;
225        do {
226            unsigned char *virt_addr = bh->b_data;
227            unsigned int size = bh->b_size;
228
229            while ((bh = bh->b_reqnext) != NULL) {
230                if ((virt_addr + size) != (unsigned char *) bh->b_data)
231                    break;
232                size += bh->b_size;
233            }
234            memset(&sg[nents], 0, sizeof(*sg));
235            sg[nents].address = virt_addr;
236            sg[nents].length = size;
237            nents++;
238        } while (bh != NULL);
239
240        return pci_map_sg(hwif->pci_dev, sg, nents, hwif->sg_dma_direction);
241    }
```

这里的函数 pci_map_sg( )实际上并没有什么操作，只是对参数的合理性作一些检验，然后返回 nents，即区间的个数。这样，操作请求所涉及的一串缓冲区就合并成了若干区间，由一个 scatterlist 结构数组代表。数组的大小为 nents，最后保存在 ide_hwif_t 结构的 sg_nents 字段中。不过，这个 scatterlist 结构数组还不是 DMA 区间表，所以在回到 ide_build_dmatable( )中(260 行)还要据此建立起 DMA 区间表。

DMA 区间表也是以缓冲区间为基础的。但是，由于 IDE 接口中 DMA 控制部分的设计，缓冲区间不能跨越 64KB 的边界(显然，用来发出 32 位内存地址的寄存器分成两截，其高 16 位在整个 DMA 操作过程中固定不变)。因此，对于 scatterlist 结构数组中的每个区间，代码中通过一个 while 循环加以检查，如果跨越了 64KB 边界就要将其分割成若干 DMA 缓冲区间。与此同时，则根据最终形成的缓冲区间建立起 DMA 区间表。DMA 缓冲区间不能跨越 64KB 边界，如果缓冲区间的起点恰好与 64KB 边界对齐，则每个 DMA 缓冲区最大可达 64KB，否则就取决于起点的位置。例如，如果缓冲区的起点在 16KB 处，那么最大可达 48KB(见 285 行)。DMA 区间表是个 32 位无符号整数数组，数组中每两个无符号整数合在一起描述一个 DMA 缓冲区，称为“物理区间描述”(Physical Region Descriptor)。其中第一个无符号整数为缓冲区起点的物理地址(289 行)，第二个为缓冲区的长度(293 行)，均须转换成“Little Ending”格式。区间的起点在一开始(267 行)就通过宏操作 sg_dma_address( )转换成了物理地址，所以随后计算出来(294 行)的都是物理地址。宏操作 sg_dma_address( )的定义在 include/asm-i386/pci.h 中：

```
164     #define sg_dma_address(sg)   (virt_to_bus((sg)->address))
165     #define sg_dma_len(sg)       ((sg)->length)
```

总线地址实际上就是物理地址，宏操作 virt_to_bus 也就是 virt_to_phys，定义于 include/asm-i386/io.h：

```
157     /*
158      * IO bus memory addresses are also 1:1 with the physical address
159      */
160     #define virt_to_bus virt_to_phys
161     #define bus_to_virt phys_to_virt
```

为 DMA 区间表分配的空间是一个页面，分用于 IDE 接口的两个设备，所以每个操作表的容量为 PRD_ENTRIES，即 256。如果要求一次就读/写超过 256 个 DMA 缓冲区(难以想像)，那就不能通过 DMA 操作来完成了，此时 ide_build_dmatable( )返回 0，从而使 ide_dmaproc( )返回 1(见 488 行)。从 do_rw_disk( )的代码中(425 和 435 行)可以看出，当 ide_dmaproc( )返回 1 时，就又回到程序控制 I/O 方式的代码中继续执行，仍能完成所要求的读/写，只不过效率低一些而已。

回到 ide_dmaproc( )的代码中(487 行)，准备好 DMA 区间表以后，下面就是对 DMA 控制器中有关寄存器的操作了。为便于阅读，我们再列出这个函数中的关键几行：

[do_rw_disk( ) > ide_dmaproc( )]

```
489                 outl(hwif->dmatable_dma, dma_base + 4); /* PRD table */
490                 outb(reading, dma_base);                /* specify r/w */
```

```
491                 outb(inb(dma_base+2)|6, dma_base+2);/* clear INTR & ERROR flags */
492                 drive->waiting_for_dma = 1;
493                 if (drive->media != ide_disk)
494                     return 0;
495                 ide_set_handler(drive, &ide_dma_intr, WAIT_CMD,
                                    dma_timer_expiry); /* issue cmd to drive */
497                 OUT_BYTE(reading ? WIN_READDMA : WIN_WRITEDMA, IDE_COMMAND_REG);
497             case ide_dma_begin:
498                 /* Note that this is done *after* the cmd has
499                  * been issued to the drive, as per the BM-IDE spec.
500                  * The Promise Ultra33 doesn't work correctly when
501                  * we do this part before issuing the drive cmd.
502                  */
503                 outb(inb(dma_base)|1, dma_base);        /* start DMA */
504                 return 0;
```

首先将 DMA 区间表的(物理)地址 hwif->dmatable_dma 写入描述表指针寄存器(489 行)。把操作的方向(读或写)写入其“命令寄存器”(490 行)，并将状态寄存器中的“中断请求”和“出错”两个标志位清成 0(491 行)。IDE 接口的 DMA 控制部分共有 16 个字节，前 8 个用于接口上的第一个设备 ide0，后 8 个则用于 ide1。根据具体的设备，初始化时就把相应 ide_hwif_t 数据结构中的字段 dma_base 设置成指向其 DMA 控制部分的起点，这就是这里的 dma_base（见 466 行）。其中从地址 dma_base 开始是 8 位的命令寄存器，从(dma_base+2)开始是 8 位的状态寄存器，从(dma_base+4)开始则是 32 位的描述表指针寄存器。

然后，通过 ide_set_handler( )设置好本次 DMA 操作结束时的中断服务程序 ide_dma_intr( )，同时也设置好超时处理程序 dma_timer_expiry( )，ide_set_handler( )的代码在 drivers/ide/ide.c 中：

```
532     void ide_set_handler (ide_drive_t *drive, ide_handler_t *handler,
533                         unsigned int timeout, ide_expiry_t *expiry)
534     {
535         unsigned long flags;
536         ide_hwgroup_t *hwgroup = HWGROUP(drive);
537
538         spin_lock_irqsave(&io_request_lock, flags);
539         if (hwgroup->handler != NULL) {
540             printk("%s: ide_set_handler: handler not null; old=%p, new=%p\n",
541                 drive->name, hwgroup->handler, handler);
542         }
543         hwgroup->handler       = handler;
544         hwgroup->expiry        = expiry;
545         hwgroup->timer.expires  = jiffies + timeout;
546         add_timer(&hwgroup->timer);
547         spin_unlock_irqrestore(&io_request_lock, flags);
548     }
```

最后，向 IDE 硬盘的命令寄存器发出 WIN_READDMA 或 WIN_WRITEDMA，启动硬盘的操作(496

行)，紧接着(503 行)向其 DMA 命令寄存器也发出启动命令(将其最低位设成 1)。此后，IDE 硬盘和 DMA 控制器的操作对于 CPU 就是透明的了。以读操作为例，IDE 硬盘在有了供读出的数据以后便向其 DMA 控制部分发出电信号。而 DMA 控制部分，则从 DMA 区间表中找到第一个缓冲区间。然后产生出反复从 IDE 接口读出数据并写入缓冲区间所需的电信号。如果一个缓冲区间满了，就从 DMA 区间表中找到下一个缓冲区间，再继续往里写。这样，一直要到完成了当前的整个 DMA 操作以后才会向 CPU 发出一个中断请求，使 CPU 转入 DMA 操作的中断服务程序 ide_dma_intr( )中，这个函数的代码在 drivers/ide/ide-dma.c 中：

```
189     /*
190      * dma_intr( ) is the handler for disk read/write DMA interrupts
191      */
192     ide_startstop_t ide_dma_intr (ide_drive_t *drive)
193     {
194         int i;
195         byte stat, dma_stat;
196
197         dma_stat = HWIF(drive)->dmaproc(ide_dma_end, drive);
198         stat = GET_STAT( );          /* get drive status */
199         if (OK_STAT(stat,DRIVE_READY,drive->bad_wstat|DRQ_STAT)) {
200             if (!dma_stat) {
201                 struct request *rq = HWGROUP(drive)->rq;
202                 rq = HWGROUP(drive)->rq;
203                 for (i = rq->nr_sectors; i > 0;) {
204                     i -= rq->current_nr_sectors;
205                     ide_end_request(1, HWGROUP(drive));
206                 }
207                 return ide_stopped;
208             }
209             printk("%s: dma_intr: bad DMA status\n", drive->name);
210         }
211         return ide_error(drive, "dma_intr", stat);
212     }
```

DMA 中断的发生意味着一次 DMA 操作的结束，因此以功能码 ide_dma_end 为参数调用 ide_dmaproc( )。同样，为方便阅读，我们再把这个函数中有关的片断列出于下：

[ide_dma_intr( ) > ide_dmaproc( )]

```
505             case ide_dma_end: /* returns 1 on error, 0 otherwise */
506                 drive->waiting_for_dma = 0;
507                 outb(inb(dma_base)&~1, dma_base);   /* stop DMA */
508                 dma_stat = inb(dma_base+2);      /* get DMA status */
509                 outb(dma_stat|6, dma_base+2); /* clear the INTR & ERROR bits */
510                 ide_destroy_dmatable(drive);     /* purge DMA mappings */
511                 return (dma_stat & 7) != 4; /* verify good DMA status */
```

首先将 DMA 控制器命令寄存器的最低位清 0，使其停止运行；然后读入其状态寄存器，并将其中断标志位 INTR 和出错标志位清 0。这里还调用了一个函数 ide_destroy_dmatable( )，其代码在 drivers/ide/ide-dma.c 中：

[ide_dma_intr( ) > ide_dmaproc( ) > ide_destroy_dmatable( )]

```
311    /* Teardown mappings after DMA has completed.  */
312    void ide_destroy_dmatable (ide_drive_t *drive)
313    {
314        struct pci_dev *dev = HWIF(drive)->pci_dev;
315        struct scatterlist *sg = HWIF(drive)->sg table;
316        int nents = HWIF(drive)->sg_nents;
317
318        pci_unmap_sg(dev, sg, nents, HWIF(drive)->sg_dma_direction);
319    }
```

这里的 pci_unmap_sg( )实际上并没有什么操作。其实，DMA 区间表也无须废除，用于 DMA 区间表的页面并不需要释放，下一次要进行 DMA 操作时自会再在同一个页面中建立起新的 DMA 区间表。

然后，如果状态寄存器指示操作并未出错，则对操作请求中涉及的缓冲区调用 ide_end_request( )。这个函数的代码已经在前面讲述程序控制 I/O 时读过了，读者可以回过去复习一下。另一方面，这也说明 DMA 操作和程序控制 I/O 殊途同归，都到了调用 ide_end_request( )的时候，下面就都一样了。

上面我们以 IDE 硬盘为例说明了块设备的驱动。实际的块设备种类当然很多，光是 IDE 设备就有硬盘、软盘、磁带、光盘（CD_ROM）及可写光盘等。我们不可能在一本书中一一加以介绍。但是，只要理解了对 IDE 硬盘的驱动，读者在进一步阅读分析其他设备的驱动程序代码时就不至于有太大的问题了。此外，除 IDE 硬盘外，SCSI 硬盘也是很常用的，但是限于篇幅我们也不能对 SCSI 总线以及有关设备的驱动再作介绍了。有兴趣或需要的读者可以参考有关专著或技术资料，自行阅读有关的代码，这些代码大多在 drivers/scsi 目录下。

最后，还有一个非常值得一提的话题是磁盘阵列。在 drivers/md 目录下有 raid0.c、raid1.c 等文件是关于磁盘阵列的，但是限于篇幅也不能在本书中加以介绍，而只好留给读者了。读者也许对此感到惋惜，我们也感到遗憾。

8.6 字符设备驱动概述

在内核中，字符设备的驱动是最多样、最灵活多变的部分。这首先是因为字符设备本身的多样性，各种字符设备在作用、功能、结构等等方面真是五花八门。有的“字符设备”甚至并不是字面意义上的“设备”，如下面要讲到的/dev/null 就是一个例子；而有的则又相当复杂而需要把驱动程序进一步划分成若干子层，或者进一步分解成若干项低层的设备，如 PC 机的控制台终端实际上就包括了显示器和键盘。其次，这种多样性还来自对设计和开发字符设备驱动程序的广泛参与。全世界通过英特网参与 Linux 开发的人数以千计，如果说如进程、调度、进程间通信、内存管理等方面的工作相对而言还

比较集中于某一些核心人士的话，那么对于设备驱动，特别是字符设备驱动方面的工作就分布得很广了。这种广泛的参与当然会带来一些风格上、技巧运用上、程序结构上的多样性。不过，由于 Linux 内核总体上的结构性和模块性，这种多样性只是次要的，并不占主导地位。最后，多样性还来自不同的历史渊源。其中最重要的是对一些（并非全部）网络设备（如 Ethernet 接口卡）的驱动，这些设备的驱动从一开始就纳入了“插口”（socket）机制的范畴而并不遵循 Unix/Linux 为设备驱动设计的统一的格局，那就是通过主设备号/次设备号来区分设备并且引导 CPU 进入具体设备的驱动程序。因此，这些设备似乎既不属于块设备，也不属于字符设备，其驱动方式自成一格，并且没有相应的“设备文件”。由于篇幅的限制，本书将不涉及网络方面的内容，而只专注于传统的设备驱动模式。尽管有些网络设备的驱动实际上确实遵循这种统一的格局，我们也只好从略。

正如我们反复讲过的那样，传统的、统一的 Unix 设备驱动是以主/次设备号为纲的。每项设备都属于块设备或字符设备，都有惟一的主设备号和次设备号，而内核中则有块设备表和字符设备表，根据设备的类型和主设备号便可通过这两个设备表之一找到相应的驱动函数跳转结构，而次设备号则一般只用作同类型设备中具体设备项的编号（通常决定着接口的 I/O 地址）。但是，由于字符设备的多样性，有时候也用次设备号作进一步的归类。这方面典型的例子就是终端设备 TTY。TTY 设备是字符设备，主设备号为 4，但是当次设备号为 0 时表示“当前虚拟控制终端”，而 1～63 表示 63 个可能的“虚拟控制终端”，64～255 则表示 192 个可能的串行口 UART(通用异步收发器)和连接在上面的实际终端设备。这里所谓实际终端设备通常是指老式的 CRT 终端，或“笨终端”，而“虚拟控制终端”则通常是建立在 PC 机的显示器和图形接口卡基础上的。显然，在这里相同的主设备号并不意味着相同的驱动程序。由于控制终端的重要性和复杂性，我们将用两节的篇幅专门介绍它的驱动程序。另一个例子是主设备号为 1 的“字符设备”，当次设备号为 1 时表示物理内存/dev/mem，为 2 时则表示内核的虚存空间/dev/kmem，为 3 时就表示“空设备”/dev/null，而次设备号 8 则表示随机数生成器/dev/random。类似的情况在块设备中虽然也有（例如对软盘设备），但是很少。随同 Linux 内核发布的一个文件 Documentation/devices.txt 列举了对块设备和字符设备两种主设备号和次设备号的分配和指定，读者可以参阅。

以前我们还讲过，每项设备都有一个代表着它的设备文件。但是，当一项设备的驱动划分成若干层次而形成一个所谓设备驱动层“堆栈”，尤其是当通过若干个可安装模块实现时，就引发了一个问题：这项设备由几个设备文件代表？是每一子层都有一个文件还是整个“堆栈”只由一个文件代表？我们知道，所谓一个设备文件代表一项设备，实质上是代表一个驱动程序的入口以及与之相联系的数据结构。用“面向目标程序设计”的话来说，就是代表着一个目标。所以，一个具体的子层（模块）是否有相应的设备文件取决于是否构成意义上独立的“设备”。在“可安装模块”一节里所引的例子中，低层模块一方面向其上层登记，另一方面又在文件系统中创建起相应的设备文件节点，就是因为这个模块本身就构成一项独立的设备，应用程序有可能需要绕过其高层直接进行读/写。另一方面，当从一个层次进入下一个层次时，通常意味着一次选择，因此需要加以引导。以上面提到的“虚拟终端”为例，当从虚拟终端层进入“物理终端”层时，就意味着一次选择，或者说“转接”：这个物理终端是接在某个串行口 UART 上的“笨终端”（或者模拟笨终端的计算机）呢？还是接在 VGA 卡上的显示器？我们在块设备驱动一节中看到了，在类似的情况下是由主设备号/次设备号引导的，但那并不是惟一的方法。从上层进入下层的路径既可以临时加以选择（例如根据设备号），也可以预先设置好。这种设置可以在系统（或设备）初始化时进行，也可以通过对高层的 ioctl( )操作随时进行，还可以是

程序中固有的。在以后几节中，读者将通过阅读代码看到这些技巧的运用。

在这一节中，我们将阅读几个简单字符设备驱动程序的源代码，这些设备都是比较简单的,以后还将阅读虚拟终端驱动程序的代码，那就比较复杂了。由于打开设备文件的过程都大致相同，我们就不再重复，而直接从设备的 file_operations 数据结构开始。

我们要看的第一个字符设备，其实谈不上是“设备”，但是却很常用，那就是“空设备”，即/dev/null。大家知道，应用程序在运行的过程中一般都要通过其预先打开的标准输出通道或标准出错信息通道在终端显示屏上输出一些信息，但是有时候（特别是在批处理中）不宜在显示屏上显示这些信息，又不宜将这些信息重定向到一个磁盘文件中，而要求直接使这些信息流入“下水道”而消失，这时候就可以用/dev/null 来起这个“下水道”的作用。如前所述，这个设备的主设备号为 1。主设备号为 1 的设备其实并不是“设备”，而都是与内存有关，或者在内存中（不必通过外设）就可以提供的功能，所以其符号为 MEM_MAJOR，定义见 major.h:

```
19      #define MEM_MAJOR     1
```

其 file_operations 结构为 memory_fops，定义见 drivers/char/mem.c:

```
613     static struct file_operations memory_fops = {
614         open:         memory_open,     /* just a selector for the real open */
615     };
```

但是，如前所述，主设备号为 1 的字符设备需要根据次设备号进一步区分具体的设备驱动程序，所以 memory_fops 还不是最终的 file_operations 数据结构，还需要由 memory_open( )进一步加以确定和设置，其代码在同一文件(mem.c)中：

```
549     static int memory_open(struct inode * inode, struct file * filp)
550     {
551         switch (MINOR(inode->i_rdev)) {
552             case 1:
553                 filp->f_op = &mem_fops;
554                 break;
555             case 2:
556                 filp->f_op = &kmem_fops;
557                 break;
558             case 3:
559                 filp->f_op = &null_fops;
560                 break;
561     #if !defined(__mc68000__)
562             case 4:
563                 filp->f_op = &port_fops;
564                 break;
565     #endif
566             case 5:
567                 filp->f_op = &zero_fops;
568                 break;
```

```
569             case 7:
570                 filp->f_op = &full_fops;
571                 break;
572             case 8:
573                 filp->f_op = &random_fops;
574                 break;
575             case 9:
576                 filp->f_op = &urandom_fops;
577                 break;
578             default:
579                 return -ENXIO;
580         }
581         if (filp->f_op && filp->f_op->open)
582             return filp->f_op->open(inode,filp);
583         return 0;
584     }
```

因为/dev/null 的次设备号为 3，所以其最终的 file_operations 数据结构为 null_fops，仍定义于 mem.c:

```
521     static struct file_operations null_fops = {
522         llseek:      null_lseek,
523         read:        read_null,
524         write:       write_null,
525     };
```

由于这个结构中的函数指针 open 是 NULL，在打开文件时没有任何附加操作。当通过 write( )系统调用写这个文件时，相应的驱动函数为 write_null( )，这个函数的代码也在 mem.c 中：

```
344     static ssize_t write_null(struct file * file, const char * buf,
345                         size_t count, loff_t *ppos)
346     {
347         return count;
348     }
```

就是说，它什么也不干，而只是返回 count，假装要求写入的那么多字节都已经写好了，实际的效果就是把要写的内容都丢弃了(读者也许会联想到某些公务员的行为特征)！

那么，通过系统调用 read( )的读操作又如何？再看 read_null( )的代码：

```
338     static ssize_t read_null(struct file * file, char * buf,
339                         size_t count, loff_t *ppos)
340     {
341         return 0;
342     }
```

返回 0 表示从这个文件中读了 0 个字节，但是并未到达（永远不会到达）文件的末尾 EOF，即－1。当然，字符设备的驱动不会都这么简单，但是总的框架是一样的。

8.7 终端设备与汉字信息处理

每个受控制的系统都要有个“控制台”（console），主要用于系统的引导、控制和管理。最原始的控制台是一些开关之类的东西，后来改成了电动打字机（TTY 就是电动打字机的意思），再后来又改成了 CRT 终端，即带显象管的终端，大多是不带智能的所谓“笨终端”。另一方面，系统的用户，特别是在多用户系统中，也需要有个作为人机交互手段的独立的终端设备，一般也都采用 CRT 终端。特别地，用户向系统登录时使用的终端就称为该用户的“控制终端”。在早期的应用中，这些终端一般都是面向文字的，特别是面向拼音文字的（如英文、俄文等），一般都没有图形/图像功能，通常也不要求很强的脱机编辑功能，所以在终端设备中一般不需要有微处理器，或者说不带智能，所以称为“笨终端”。严格说来，纯粹作为控制台使用的设备与作为控制终端使用的设备在功能要求上有所不同，但是相比之下这些不同一般都不大，由此而造成的设备价格上的不同也并不突出，所以人们常常倾向于采用同类的终端设备，而把其中一台保留给“系统管理员”用作控制台。

随着计算机应用的普及和发展，在有些应用中对终端设备有了一些特殊的要求，例如对图形/图像的支持、对脱机编辑功能的要求等等，而其中最重要的则莫过于对非拼音文字如汉字、日文等等的支持。这些终端设备的功能都比较复杂，所以本身都带有微处理器而成为“智能终端”。80 年代初期人们常常看到的“汉字终端”就是支持汉字输入和显示的智能终端，但是智能终端的出现并不给操作系统内核的设计和实现带来显著的影响。以汉字终端为例，内核只要不排斥 16 位的汉字编码就可以了(一般而言，Unix 内核并不排斥汉字编码）。具体到对汉字的输入、显示、打印和各种处理都是由汉字终端、中文打印机以及各种中文应用软件在操作系统以外提供支持的。那时候，典型的计算机系统形式就是一台“主机”带上许多终端，其中一台是由系统管理员使用的“控制台”。

个人计算机（PC）的出现和发展使情况发生了显著的改变。在 PC 机上，人们通过显示器和键盘（通常还有鼠标器）进行人机交互。这些设备合在一起既用作系统的控制台又是用户的控制终端，而且与“主机”融合在了一起，CPU 的一部分“能力”就用于显示器和键盘的驱动。这样，原来独立存在于终端设备中的一些功能，例如对文字显示的支持，就转移到了系统的内核中。以英文字母的显示为例，在以前采用笨终端的 Unix 系统中，内核只要把代表着字母的编码通过串行口送给控制终端（或控制台）就行了，下面就是具体终端的事了，主机对此既不需关心也鞭长莫及。而现在却不同了，内核可能要管到屏面上的每个像素为止，包括根据具体字母的代码和字体找到代表着该字母的“字模”点阵，然后将该点阵中的每一点都映射到屏面上的一个像素。显然，这就相当于在原先 Unix/Linux 内核对终端设备的驱动程序中又增加了一层，使驱动程序更加复杂，但是同时也使对终端设备的驱动更加灵活。例如，这么一来，对字体的选择、颜色的改变以及字母的放大/缩小等等就比较容易实现了。而特别重要的是，对非英文字母以及非拼音文字的支持就基本上可以不依赖于硬件了。但是，另一方面，由显示器和键盘所构成的终端设备并不是 Linux 内核所支持的惟一终端设备。只要在 PC 机的串行口（COM1/COM2）上插上笨终端（或者用来模拟笨终端的其他 PC 机），就照样可以把运行着 Linux 的 PC 机用于多用户环境。进一步，如果在 PC 机上加上足够的串行接口，就照样可以带上几十个甚至上百个终端而成为名副其实的“主机”。当然，对连在串行口上的终端的驱动不同于对显示器和键盘的驱动。Linux 内核对汉字输入和显示的支持只限于其控制台，即显示器和键盘。也就是说，如果要在

除显示器和键盘以外的用户界面上（终端上）支持汉字，那么这些终端或用户界面本身就得带有智能而支持汉字的输入和显示，例如汉字终端，或者在其他 PC 机上运行的汉字终端仿真软件以及（在 web 环境下）支持汉字的浏览器等。不过，技术发展的趋向是连成网络的 PC 机(或“工作站”)，所以实际上已经很少有对汉字终端的需求了。

最后，还有一个问题也是要考虑的，那就是与旧有软件的兼容性。一般而言，在 Unix 上开发的 C 语言程序拿到 Linux 上重新编译以后就应该可以运行，而“Shell script”（Shell 过程）则应该直接（最多略加修改）就可以执行。这样，就不允许对设备驱动的用户界面作太大的改动，例如不能将由显示器和键盘构成的终端与普通终端通过不同的主设备号加以区分，否则碰到类似“echo please wait >> /dev/tty0”的语句就不能执行了。可见，将 Unix 内核的终端设备驱动加以改造，使之既适合由显示器和终端构成的特殊终端设备，又适合普通的、传统的终端设备，还要考虑到兼容性，确是一项富有挑战性的任务，而对多种文字的支持，则使它更加复杂了。

在迄今为止的计算机发展史上，大部分的技术和应用都首先出现在美国和欧洲，因而最初都是以英文为载体的。英文是拼音文字，而且其字母表又很小，连大小写字母加数字加常用符号全都算上还不到 127 个，所以英文（美国英文）的编码方案 ASCII 是 7 位的，这又正好与初期的 RS232 串行接口每次只能传输 7 位（另一位用作检错）相符。虽然当时在计算机中已经广泛采用以 8 位为一个字节，在用来存储字符时则固定使最高位为 0。后来，由于考虑到一些特殊符号，如一些数学符号和常用的希腊字母，再说 RS232 串行接口上的传输也变得可靠而不再需要对传输的每个字符都加上奇偶检验，才把 ASCII 编码扩充成 8 位。然而，当要把计算机技术和应用推广（或者销售）到其他国家和地区时，不可避免地会碰到一个问题，那就是文字“本土化”的问题，即对所在国家或地区的文字的支持问题。世界上多数国家都使用拼音文字，尽管其字母表有大有小，但是 8 位的编码可以容纳 256 个代码，一般都够了，而且恰好字节的大小也是 8 位，RS232 串行口的传输也是每次 8 位，所以用一个字节表示一个字符就成了事实上的国际标准。但是，还有些国家和地区使用的是非拼音文字，其中最主要的就是所谓 CJKV，即中文、日文、朝鲜文和越南文。可想而知，对于像中文这样的非拼音文字，要继续以一个字节表示一个字是不可能的，因为一个字节总共才有 256 种不同的编码。但是到底以几个字节表示一个汉字，每个字节中用 7 位还是 8 位，是固定长度还是可变长度，怎样编码，在一个字符串中是否允许混合使用单字节编码和多字节编码，以及怎样在二者之间切换等等，则都是值得研究的课题。事实上，在不同的地区、不同的条件下已经发展起了多种不同的编码方案。

很自然地，将各种文字的编码方案加以标准化和一体化的努力也就应运而生。这里所谓“标准化”是指国际标准或跨国行业标准的制订，而一体化则是指将世界上所有的文字都纳入同一编码方案（以及标准）里面。回顾这方面的历史，下面几个标准是必须提到的：

(1) 美国标准 ASCII。

(2) 国际标准 ISO646。

(3) 国际标准 ISO8859。

(4) 国际标准 ISO2022 和中国国标 7 位 GB2312。

(5) Unix 行业标准 EUC 和中国国标 8 位 GB2312 及 GBK。

(6) 计算机及信息行业标准 Unicode。

(7) 国际标准 ISO10646。

先要说明，我们在这里只是为阅读和理解 Linux 内核代码以及汉化 Linux 内核的需要而介绍一些背

景材料，采取的是“带着问题学”的实用主义态度。读者要了解详细的、比较完整的材料应参阅有关专著以及这些标准的文本。

美国的国家标准 ASCII（美国标准信息交换码）公布于 70 年代前期。这个标准把数据交换采用的编码定为 7 位。这一方面是由于当时通过串行接口 RS232 的数据传输还不太可靠，需要对每个代码使用一位作为奇偶校验，而且当时传输的速度也比较低，能够在每个代码中节省下一位就可以使总体的吞吐量增加 10%左右。再说，当时人们的头脑中可能觉得 7 位已是不简单了，在此之前还使用过 5 位和 6 位编码呢。另一方面，当时的人们也觉得超过 7 位是没有必要的，因为 7 位编码的容量是 128，而美国英语中的可打印字符才 94 个（包括空格在内）。至于非打印字符，除了作为控制字符使用外，当时的人们也看不出在数据交换中有多大用处。当时的计算机内部结构可说是五彩缤纷，有的厂家采用 8 位，有的厂家采用 10 位，数据的内部表示也由厂家自己定义（IBM 就采用 8 位 EBCDIC 编码），所以除了采用划一的“打印模式”外，确实也看不到有“透明”地传输计算机内部数据的必要性和可行性。随着计算机技术的发展，ASCII 自然地变成了事实上的国际标准。一个标准一旦制订发布，各种软件就会以此为准来设计和开发。这样，当过了若干年以后，原先考虑的因素都已经因为技术的进展而不复存在的时候，一大批以此为基础的软件却已经在那里运行，以后开发的软件只好继续采用 7 位编码以维持兼容，从而形成了本不应存在的壁垒。更有甚者，在有些特殊应用中甚至还从 95 个打印字符中“克扣”了一些字符用作控制目的，使可以“透明”传输的字符个数进一步减少。建立在 64 个可以“透明”传递的可打印字符基础上的电子邮件编码方案 Base64，就是一个典型的例子。

在 ASCII 成为事实上的国际标准以后，国际标准化组织 ISO 通过其 ISO646 加以确认，使其成为正式的国际标准“参考版”之一。同时，又从中提出若干如[、]、{、}、\ 和 | 等符号的代码，让字母个数多于 26 个的语言可以用这些符号的代码来表示一些在英语中不存在的字母，如 æ 等等。但是，那已经是 80 年代前期的事了，那时 7 位的传输技术实际上早已过时。正因为这样，ISO 从 80 年代中期开始陆续推出了另一个标准 ISO8859，这个标准采用 8 位传输。ISO8859 分成 10 个部分，为世界上存在的几乎所有拼音文字如何进行 8 位编码都作了规定。例如其第 6 部分，即 ISO8859-6 是阿拉伯文字的编码，第 7 部分是希腊文字的编码，第 8 部分是希伯来文字的编码。对于拼音文字来说，8 位的编码容量都已经够用了。

但是，ISO8859 并没有解决一体化的问题，同一个 8 位代码对于不同的语言文字就代表着不同的字母，在混合使用多种文字的上下文中要加以切换。另一方面，新标准的制订和采用并不意味着旧的标准就退出了舞台。大量基于旧标准的软件已经在使用中，当要开发新产品的时候，虽然明知道新的标准已经发布，但为了不至失去一大块市场，只好从最坏、最严苛的条件作为出发点，即继续采用 7 位编码，或至少要提供用户一个可选项，让用户根据具体情况来设置到底是采用 7 位编码还是 8 位编码。

这几个标准都只考虑了拼音文字，所以都采用了单字节编码。与此相平行，一些采用非拼音文字的国家和地区也已开始为适合其语言文字的编码制订国家标准或行业标准。其中最早的是日本，在 1978 年初发布了 JIS C 6626 标准，这个标准中的汉字部分后来成为台湾地区制订“大五”编码的借鉴。中国在这方面的工作也开展得很早，1980 年就发布了国标 GB2312。当然，这些标准都采用了多字节编码，大多是双字节编码。

ISO2022 是第一个支持多字节编码，即非拼音文字的国际标准，发布于 90 年代前期。事实上，这又只不过是对有关国家和地区在这方面的实践的事后追认和兼容并包而已。所以，ISO2022-CN 与中国

的 GB2312 很接近，ISO2022-JP 与日本的 JIS C 6626 很接近，而 ISO2022-KR 则与韩国的标准很接近。ISO2022 保留对英文字母的单字节 ASCII 编码，当使用 CJK（中/日/韩）文字时就转入双字节编码，两个字节都采用 7 位，取值范围为 0x21～0x7E，即"可打印字符"的范围。为了保证在不同字符集之间正确切换，ISO2022 定义了一些"Escape 序列"，称为"标示序列"（designator sequence）。与键盘上的 ESC 键相对应的代码为 0x1b，这并不属于可打印字符的范围，在传输中平时不会用到，所以在计算机技术中与终端或传输线打交道时常常用 ESC 的代码作为控制手段，与其他字符组合成 Esccape 序列。此外，还要保证在单字节模式与双字节模式之间正确切换，ISO2022 为此定义了两个控制字符。一个是 0x0e，称为"SO"（Shift Out），表示转出（常规的）单字节模式；另一个是 0x0f，称为"SI"(Shift In)，表示转入单字节模式。这样，当行文至需要使用非拼音文字时，就先通过 SO 进入双字节模式，再通过 Escape 序列选择具体的字符集，例如当目标为 GB2312 时，其序列为"0x1b, 0x24, 0x29, 0x41"或"<ESC>$)A"。然后，当要转回单字节模式时，就在字符串中下一个英文字母之前插入一个 SI 代码即可。

这种技术的好处是效率比较高，从传输的角度看浪费很少，但是从信息处理的角度看却有缺点，因为它是面向过程的，要知道对某一个具体的字节应作何种解释取决于这个字节在传输时处于何种模式，所以对于字符串（甚至整个文件）中的内容只能顺序访问而不能随机访问，因为对每个字节的解读都是与上下文有关的。

另一方面，在某些特殊的应用中，特别是电子邮件的传递中，这套技术还是有问题。前面提到过，电子邮件的传递只保证 64 个可打印字符的透明传输，而 0x0e、0x0f 和 0xlb 显然不属于这个范围。对此，有一个中国留美学者提出了采用另一种序列来实现 ASCII 码与 GB2312 码之间的切换，称为 HZ 编码（RFC 1834）。当要从单字节的 ASCII 码切换到双字节的 GB2312 码时，就插入"～{"，要回到 ASCLL 码时则插入"～}"。这些字符都属于可打印字符。而"～}"也不对应于任何汉字的编码，显然，这里的关键是赋予字符"～"以特殊的解释，所以当在单字节模式下真要使用这个字符时，就要在它前面再添一个"～"字符。读者肯定会想到，这与 C 语言中"\"字符的使用在精神上是一致的。

那么，Linux 的内核是否支持 ISO2022（从而 GB2312）呢？前面讲过，这要分两种情况来考虑。第一种情况是如果采用带有智能的终端设备，即支持 GB2312 的汉字终端，此时只要内核不排斥 GB2312 的编码就行了。第二种情况是当内核需要介入汉字的输入和显示，那又是另一个问题了。

先看第一种情况，Linux 的内核是否排斥 ISO2022 呢？这是个字符串处理的问题，主要关系到字符串的界定、比对，以及一些特殊字符的使用。内核中字符串都是以"\0"结尾的，所以如果在双字节编码中有某个汉字的代码包含了一个 0 字节，那就会被内核误认为是字符串的结尾。可是，ISO2022 中两个字节取值的范围都是 0x21～0x7e，因而不会出现这样的情况。同理，在汉字编码中也不会出现像^C、^D、^Z 以及"退格"这些控制字符，也不会有 0xff（EOF 定义为－1）。再看字符串的比对。内核中要用到字符串比对主要是在文件系统的路径名搜索。在包含汉字的路径名中，需要把换码字符 SI/SO 作为每个节点名字符串的一部分，对于像 Linux 这样对文件名长度并无过分限制的操作系统来说，问题似乎也不大。可是，对另一个特殊字符的解释就成问题了，这个字符是"/"。例如，假定有这样一个路径名："/usr/students/classes/化工 98"，这就造成问题了。这是因为汉字"化"的编码中含有 0x2f，也就是"/"。读者在文件系统一章中阅读过 path_walk( )的代码，在那里这个字符一定会被当作节点名的分隔符而表示另一个层次。这样的汉字当然有很多。所以，结论是：只要不使用汉字作为文件名或目录名，Linux 内核便不排斥汉字的使用。同样的结论也适用于 Unix。对于 80 年代准备

要带上几十个汉字终端的 Unix 系统来说，这个结论已经是足够令人满意的了。

不管是多么权威的标准，只要是束缚了人们的手脚，一旦客观的、物理的限制因素（如硬件的可靠性等）消除以后就一定会被冲破。但是，冲破已经被广泛接受的标准往往要付出不小的代价，这就是新产品的部分市场或适用范围。可要是这市场本来就不存在呢？那自然又作别论。80 年代正是 Unix 气势如虹，从传统的小型机、中型机乃至大型机的手中攻城掠地的时候。当时，硬件（RS232 接口）方面对采用 7 位通信方式的限制已经不存在，而 Unix 机通信的对象一般也都是 Unix 机，不存在已经有大量旧软件在运行而必须保持兼容的问题。在这样的情况下，标准就失去了权威。为了将 Unix 推行到采用非拼音文字的国家和地区，采用了一种称为 EUC（扩充 Unix 编码）的 8 位多样编码方法。EUC 也像 ISO2022 一样混合使用单字节和多字节（一般是双字节）编码。单字节编码也是 7 位的 ASCII 码，也就是说每个字节的最高位永远是 0；但是多字节编码则采用 8 位，把其中某几个字节的最高位设置成 1，因而不使用换码控制字符就可以达到在单字节和多字节两种模式之间的切换。（但是也有例外，EUC 实际上允许在 4 种编码（即文字）之间进行切换，其中 ASCII 为编码 0。从 ASCII 切换到编码 1 可以通过将最高位设成 1 来实现；而若要切换到编码 2 则要插入一个控制字符 0x8e；切换到编码 3 则要插入一个 0x8f。不过在 GB2312 和 GBK 中都不考虑编码 2 和编码 3。）

因目标语言或地区的不同，EUC 编码又分成 EUC-CN（中国）、EUC-JP（日本）和 EUC-KR（韩国）、EUC-TW（中国台湾）4 种。我们在这里只关心 EUC-CN，它的基础仍旧是 GB2312，所以又常常称为“8 位国标”或干脆就称为“国标”。它采用单字节和双字节混合编码。单字节编码与 ASCII 相同，字节的最高位永远为 0；而表示汉字的双字节编码则两个字节的最高位均为 1。具体地说，两个字节的取值范围都是 0xa1～0xfe（不包括 0xff）。那么，Unix 的内核会不会排斥这些编码呢？显然不会，因为代表着汉字编码的每个字节的最高位都是 1。而且这意味着甚至可以用汉字作为文件名或目录名了。以前面讲过的“化”字为例，它的第二个字节在 7 位编码中为 0x2f，所以与“/”冲突，而现在则改成 0xaf 了。

中国后来又公布了对国标的扩充 GBK，也是采用 8 位编码，但是把第一个字节的取值范围扩充成 0x81～0xfe；把第二个字节的取值范围扩充成 0x40～0x7e 和 0x80～0xfe。显然这是为了容纳更多的汉字。虽然 GBK 编码中的第二个字节有可能最高位为 0，但不会造成什么问题，因为在 0x40～0x7e 这个区间并没有什么“敏感字符”。

EUC-CN 和 GBK 比 ISO-2022 显然有改进，但还是有缺点。首先，汉字编码的两个字节最高位均为 1（或均有可能为 1），使得在传输时容易因误码而使相邻字节的结合发生错位的情况。在这两种编码中，对每个具体字节的解释仍不是与上下文无关的（不过比 ISO2022 采用换码控制字符的编码要好多了），所以还是不能支持对字符串内容的随机访问。其次，EUC 的容量毕竟还是有限，设想如果在同一个文件中既要使用英文，又要使用中文、蒙古文、韩文，还要使用希伯来文，那又该怎么办呢？

上面提到的这些标准都没有考虑（更谈不上解决）世界文字编码一体化的问题。这些编码中没有一种是能够覆盖全世界所有常用文字的。而在受到覆盖的各国文字中，则又常常重复使用相同的代码，例如代码 0x5B 在 ASCII 中表示“[”，而在丹麦文字中却表示“Æ”，此谓“一码多字”。另一方面，有些文字却又以不同的代码重复地出现于不同的编码中，特别是中文、日文、韩文中都使用汉字，但是同一个汉字在这三种文字编码中的代码却又往往不同，从而造成“一字多码”。还有，单字节与多字节之间的切换，无论用或不用换码控制字符，都在一定程度上使对字符串中具体字节的解释依赖于上下文，从而使得对字符串内容的随机访问变得困难或甚至不可能，这同时也使得字符串在传输过程

中的抗干扰能力变差。再说，有些人可能还觉得以单字节代表英语编码，而以多字节代表其他各种文字的编码实际上是把英语置于一种中心地位，因而有失公平。

有这么多的缺点存在，人们寻求世界文字编码一体化的努力就毫不奇怪了，而 Unicode 正是这样一种一体化的编码方案，这是由一个行业协会性质的组织“Unicode Consortium”发展起来的行业标准。提到Unicode就不能不提出另一个国际标准ISO-10646，因为这二者的形成和发展是紧密结合在一起的。最初，在 80 年代前期，ISO 组成了一个工作组 WGZ 从事一体化编码方案的制订，并提出了一个框架性的方案。通常，国际标准从制订到批准、采纳是需要相当长的时间的。另一方面，作为国际标准，其视野自然更为宽阔，其结构也常常更为严谨，很多问题都不是一下子就能定得下来的。可是，当时的工业界却觉得形势逼人，不能再等，于是就组成了 Unicode Consortium，在 ISO 工作组框架的基础上以一种比较实用的态度制订行业标准 Unicode。后来，ISO 的框架性方案经过多年的发展成为国际标准 ISO 10646-1，即 ISO 10646 的第一版。目前有 ISO 10646-1:1993 和 ISO10646-1:2000 两个文本，分别发表于 1993 年和 2000 年。与此相平行，Unicode Consortium 在 1990 年便发表了 Unicode 1.0 版，以后又分别在 1996 年和 1999 年发表了 2.0 版和 3.0 版。不过，ISO 和 Unicode Consortium 在这方面的工作既互相平行，又密切结合，并且互相靠拢互相融合。1990 年 Unicode Consortium 发表了 Unicode 1.0 版以后，1991 年双方就进行了一次融合，从而产生了 1993 年的 ISO10646-1 文本。而 Unicode Consortium 又在此基础上发表了 Unicode 的 1.1 版，然后又进一步向 ISO 10646 靠拢而在 1996 年发表了 2.0 版。所以，这二者之间的关系是一种互相靠拢、互相补充、互相融合的良性互动关系。

那么，Unicode 到底是什么样的呢？这二者到底有些什么异同呢？我们在这里只是结合阅读 Linux 内核代码的需要作一简略的介绍，欲知详情的读者可以参考有关专著和文本。在这方面，有两本书是值得推荐的：一本是 Ken Lunde 的 *CJKV Information Processing*；另一本是 Tony Graham 的 *Unicode:A Primer*。

首先，ISO10646 和 Unicode 都意在将全世界所有的文字都包罗在同一个编码方案中，只是 ISO 在这方面更加追求完美，而 Unicode 则采取相对而言比较实用的态度。其次，二者都采用固定长度的字符编码。这样就摆脱了在单字节代码和多字节代码之间来回转换的问题，从而使得对字符串内容的随机访问成为可能。但是，因为 ISO 的“胃口”更大，所以 ISO10646 标准选择采用 4 字节编码 UCS-4；而 Unicode 标准则采用更为紧凑的 2 字节编码 UCS-2。以字母“a”为例，在 ASCII 编码中其代码为 0x61，而在 Unicode 中为 0x0061，在 ISO10646 中则为 0x00000061。显然，固定长度编码在许多情况下增加了对存储量以及传输带宽的要求（或者说“浪费”了一些存储空间和带宽），不像以前可变长度编码那么紧凑。但是，在计算机系统的存储空间日益加大，成本日益下降，通信能力日益上升的背景下，这一点已经是无关紧要的了。与取得的好处相比，这个代价是值得花的。如果说以前的单字节与多字节混合编码的方案与 ISO10646 的编码方案是两个极端的话，那么 Unicode 就是二者间的折衷。另一方面，虽然 Unicode 的 2 字节编码容量还是太小（例如，还不能把古埃及的文字包罗进去，更不能把中国的甲骨文字包罗进去），但毕竟已经可以覆盖全世界正在积极使用中的所有文字和符号。所以 ISO 和 Unicode Consortium 才在 1991 年对两个标准进行了融合。一方面 ISO 接受 2 字节编码的 UCS-2 为 ISO10646 的一个子集，称为“基本多语言平面”BMP（Basic Multilingual Plane），并采纳了 Unicode 中的具体代码。另一方面，Unicode Consortium 也认识到 2 字节编码的容量确实还不够，因而在 2 字节编码空间中割下一块（0xD800～0xDFFF）称为“代用码”(surrogate)，当 2 字节编码不够用时就可以采用两个“代用码”，即以 4 个字节来表示 0x10000～0x10ffff 区间的代码。那么，“代用码”的采用

会不会又破坏了对字符串内容的随机访问呢？不会，因为当采用“代用码”时位于高位的两个字节一定在 0xD800～0xDFFF 范围内，而低位的两个字节一定在 0xDC00～0xDFFF 范围内，这两块代码在正常的 UCS-2 编码中是保留不用的。由两个 2 字节代用码所表达的代码数值按以下公式计算：

$$N = 0x10000 + (H - 0xd800) \times 0x400 + (L - 0xdc00)$$

这里的 H 为高位代用码，其取值范围为 0xd800～0xdbff；L 为低位代用码，其取值范围为 0xdc00～0xdfff。

这么一来，与原先设想的 UCS-2 编码就有所不同了，所以 ISO 进一步将这种编码方案称为 UTF-16，意为过渡性的 16 位编码格式。而前述的 BMP，也就是 Unicode，实际上对应于 UTF-16，而不是原先的 UCS-2。

最后，无论是 ISO10646 或 Unicode，都排除了“一字多码”的情况，从而实现了字符（符号）与代码间的一一对应。目前，Unicode 包罗了 49194 个字符和符号的代码，其中汉字为 27786 个，这些汉字是将中文、日文和韩文中使用的汉字加以整合而成的。通过“代用码”的使用，还可以将这个容量扩大 16 倍。

可想而知，从以单字节为主，以 ASCII 码为基础的字符串处理到一律采用双字节的 Unicode 字符串处理，这是一个重大的转变，而如果还要考虑兼容性的话，那就更是个挑战。不说别的，就拿字符“a”的 Unicode 代码 0x0061 来说吧，这里面就有一个字节是 0，而传统的字符串就是以 0 结尾的。其他如 0x03(^C)、0x04(^D)、0x1b(ESC)、0x2f('/')等等字节也是随时都有可能碰到，简直是“防不胜防”。（如果可以抛开原来的单字节字符串处理不管，而在 gcc 中将数据类型 char 改成与 unsigned short 等价，那么许多现有的软件只要重新编释一下就可以支持 Unicode 了（那时候，字符串就是以 0x0000 而不是 0x00 结尾的了）。事实上，java 语言正是这样做的，在 java 语言中 char 是 16 位的而不是 8 位的。可是那样一来许多软件（包括 Linux 内核）中一些数据结构的大小就变掉了，所以问题并不简单。）

所以，指望在一个短时期内完成这种转变显然是不现实的。有鉴于此，Unicode Consortium 和 ISO 从一开始就提供了过渡性的方案。第一个过渡性方案称为 UTF-8，是从 8 位到 16 位(即双字节)的过渡方案；第二个称为 UTF-16，实际上就是 Unicode，是从 16 位到 32 位(即 4 字节)的过渡方案。

当然，我们在这里关心的只是 UFT-8，这是一种什么样的编码方案呢？简而言之，那就是单字节与多字节相结合。读者也许马上就会问：怎么绕了半天又回到单字节与多字节相结合了？我们现在的 ISO2022、GB2312、GBK 等等不正是单字节与多字节相结合吗？是的，虽然从原理上讲都是单字节与多字节相结合，但是具体的编码方法不同。UTF-8 的编码规则为：

- 7 位 ASCII 码保持原状不变。也就是数值在 0x00～0x7F 范围内的 Unicode 代码在 UTF-8 中为单字节，并且最高位为 0。
- 不能用 7 位表示，但是可以用 11 位表示，也就是数值在 0x80～0x7FF 范围内的 Unicode 代码在 UTF-8 中为双字节。第一个字节的最高三位为 110，然后是这 11 位中的高 5 位；第二个字节的最高两位为 10，然后是 11 位中的低 6 位。
- 不能用 11 位表示，但可以用 16 位表示的数值，即数值在 0x800～0xFFFF 范围内的 Unicode 代码在 UTF-8 中为三字节。第一个字节的最高四位为 1110，然后是 16 位中的最高 4 位；第二个字节的最高二位为 10，然后是 16 位中的中间 6 位；第三个字节的最高两位也是 10，然后是 16 位中的最小 6 位。
- 对于超出 16 位，但是可以用 21 位表示的数值，即数值在 0x10000～0x1FFFFF 范围内的代码，

在 Unicode 中要用两个“代用码”（surrogate）来表示，而在 UTF-8 中则为四字节编码。第一个字节的最高 5 位为 11110，然后是这 21 位中的最高三位；第二、三、四个字节的最高两位都是 10，然后各载有 21 位中的 6 位。

所以，一个字节的最高位为 0 表示这是单字节的 ASCII 码，最高位为 1 表示这是多字节代码中的一个字节，而该代码中的第一个字节的最高两位均为 1，并且连续有几位为 1 就表示该代码中有几个字节。代码中其余字节的最高两位均为 10。

这样的编码方法有什么好处呢？让我们来看看 UTF-8 编码的性质：

(1) 与 ASCII 码兼容，但是在这方面 UTF-8 并没有什么特别的优势，因为其他各种编码也都是与 ASCII 兼容的。
(2) 不使用换码控制字符，多字节代码中每个字节的最高位都是 1，并且每个多字节代码中的第一个字节很容易与其余的字节相区别，因而不存在“配对”是否正确的问题。UTF-8 编码的这个性质使对字符串内容的随机访问成为可能。当从一个字符串中随机选取一个字时，如果该字节的最高位为 0，则为单字节的 ASCII 码；如果最高两位均为 1 则为多字节代码中的第一个字节，否则往回扫描最多三个字节必可发现代码中的第一个字节。
(3) 抗干扰能力强，假定传输过程中某个字节的最高位从 0 变成了 1 或从 1 变成了 0，则受到损坏的最多只是这个字节本身所在的代码加上与它相邻的代码。而不会像在其他编码方案中那样引起成片的损坏（例如，因换码字符损坏以及因“配对”错位所造成的损坏）。
(4) 仍与 Unicode 一样，是一体化的编码。
(5) 凡是能接受 8 位码的软件就不会排斥 UTF-8 编码，因为在它的多字节代码中不含任何特殊字符（如 0x00、0x03、0x1b、0x2f 等等）。或者说，所有能接受 8 位编码的软件都是对 UTF-8 透明的。

那么，Linux 内核对 Unicode 和 UTF-8 的接受程度如何呢？我们不妨带着问题来重温一下 path_walk( )的代码。显然，path_walk( )的代码对 UTF-8 代码是透明的，因为多字节的 UTF-8 码中不含有如 0x00、0x2f 等等这些特殊字符。所以，凡是 UTF-8 编码的文字，包括汉字，都可以用在文件名和目录名中，更可以用在文件内容的任何字符串中。可是，path_walk( )对 Unicode 还是不透明的，所以 Unicode 不能用在文件名和目录名中。在这方面，对 Linux 内核还有工作要做，这也是今后 Linux 内核继续改进的方向之一。那么，退而求其次，以 Unicode 作为文件内容，就像从前的 Unix 系统带上汉字终端那样，是否可以呢？读者在阅读了有关的内核代码以后就会看到，这是可以的。

了解了上面这么些背景材料以后，我们可以回到终端设备的驱动了。

如前所述，在 PC 机上一般总是以显示器和键盘（可能还有鼠标器）的组合作为控制台的，这二者的结合就相当于一个终端。但是，有些情况下一个系统的控制台不止一个，所以 Linux 将同一套物理的显示器和键盘复用于若干“虚拟控制台”(virtual consde)，让用户通过“Alt”键与功能键“F1”至“F12”的组合来选择将某个虚拟控制台作为系统的当前控制台。由于一般键盘上有 12 个功能键，所以可以有 12 个虚拟控制台（或者虚拟终端），分别对应着设备文件/dev/tty1 至/dev/tty12。系统在初始化以后以/dev/tty1 为当前控制台。此外，/dev/tty0 永远代表着系统的当前控制台，所以如果用户按了 Alt+F2 键，则/dev/tty0 就等价于/dev/tty2。在/dev 目录中还有一个节点/dev/console，一般都是连接到/dev/tty0，所以也代表着系统的当前控制台。

这样做有什么好处呢？一来系统的用户可以在不同的虚拟终端上登录（不过 Linux 只在前面 6 个虚拟终端上创建 login 进程）并启动不同的作业，然后通过 Alt 键和功能键的组合在这些虚拟控制台之间切换，根据需要使得在某个虚拟终端上启动运行的作业成为当前的“前台作业”。另一方面，系统在引导、初始化以及运行中要显示很多信息，如果让这些信息显示在同一个虚拟终端上，则许多信息混在一起不容易看清，并且很快就被后来的信息所覆盖。如果将这些信息分门别类地显示在不同的虚拟终端上，则用户可以有选择地阅读，这样就方便多了。作为字符设备的这些虚拟终端的主设备号为 4，次设备号则与“终端号”相对应，如/dev/tty0 的次设备号为 0，/dev/tty1 的次设备号为 1，等等。

这些设备文件实际上都代表着一个输出缓冲区，写入这些设备的内容都写在缓冲区中，直到选择某个特定虚拟终端时才将其缓冲区的内容显示到显示器屏幕上。从这些设备读就不一样了，从虚拟终端读实际上是从键盘读，但是键盘只属于当前控制台，所以从虚拟终端的（同步）读操作一直要到选择了该虚拟终端作为当前控制台并从键盘输入后才会返回。

主设备号为 4 的字符设备并非全是虚拟终端，还包括了一些通过常规 UART 串行口连接的实际的终端设备。具体讲，除次设备号为 0 的/dev/tty0 代表着当前控制台，即当前虚拟终端外，次设备号 1~63 分别为/dev/tty1 至 dev/tty63,代表着 63 个虚拟终端（虽然在 PC 机键盘上只有 12 个功能键，因而只能在前 12 个虚拟终端中作出选择）；次设备号 64 至 255 则分别为/dev/ttyS0 至/dev/ttyS191，它们代表着 192 个可能的 UART 串行口，即一般的串行终端设备。

此外，与/dev/tty0 和/dev/tty1 至/dev/tty63 相对应，在/dev 目录下还有一些主设备号为 7、代表虚拟终端缓冲区的“字符设备”文件/dev/vcs 和/dev/vcs1 至/dev/vcs63。这些设备文件与代表虚拟终端的设备文件相对应，但是有所不同。从某个虚拟终端缓冲区（例如/dev/vcs2）读就是从相应虚拟终端的输出缓冲区读，而不是像虚拟终端本身那样是从键盘读。二者的写操作也有所不同，对虚拟终端缓冲区的写操作只是简单的对线性缓冲区的操作，而并不像虚拟终端那样以缓冲区来模拟显示器的屏幕。这两种设备的 file_operations 数据结构也不同，一为 vcs_fops，一为 tty_fops。

不仅如此，在/dev 目录下还有另外 64 个主设备号也是 7 的字符设备/dev/vcsa 和/dev/vcsa1 至/dev/vcsa63，它们的次设备号为 128 至 191。这些字符设备也对应着 63 个虚拟终端的缓冲区，所不同的是这些缓冲区是带有字符“属性”的缓冲区，文件名中的字母“a”就是“attribute”的意思。对 VGA、EGA 等图形卡和 BIOS 有所了解的读者也许知道，当这些图形卡工作于字符模式时可以让每个字符带上一个表示属性的字节，以控制该字符显示时的颜色、亮度等属性。

读者也许已经觉得这些终端设备不简单了。然而还不止于此，在/dev 目录下还有主设备号为 5 的字符设备。这些设备统称为“替换”（alternate）终端设备。其中次设备号 64~225 分别为/dev/cua0 至/dev/cua191，共 192 个使用串行口的终端设备，称为“callout”终端设备，与前述的/dev/ttyS0 至/dev/ttyS191 相对应。那么这二者又有什么区别呢？这就要从终端设备与主机的连接方式说起。终端设备与主机（确切地说是主机的串行口）的连接方式大体上有两种，一种是“本地”连接方式，这就是一般“计算中心”里把几十个乃至上百个终端设备直接（一般都是通过 RS232C 电缆）连接到主机上各个串行口的方式。在这种连接方式里，终端设备与主机的连接是静态的，把哪一个终端的电缆（不管有多长）插入哪一个串行口，这个终端在物理上就静态地连接到了这个串行口。终端的电源可以关闭，但是二者物理上的连接却并不改变。另一种是“远程”连接方式，在这种方式里，终端设备与主机并不直接相连，而是各自通过一个 modem 经电话线网络对接，所以终端设备与主机的连接是动态的，不同的终端设备在不同的时间里可以通过电话线网络连接到主机的同一个串行口上。然而，这里就有一个问题，

就是由谁来启动这种动态连接呢？一种办法是让主机一方启动，称为“callout”，此时的串行口工作于“呼叫”（callout）模式，相应的终端设备称为“呼出型”（callout）终端设备，即/dev/cua0 至/dev/cua191。另一种办法是让终端一方通过其 modem 拨号启动，称为“dial in”，此时的串行口工作于“拨号”模式，相应的终端设备称为“拨入型”终端设备，即/dev/ttyS0 至/dev/ttyS191。显然，这二者的驱动程序是略有不同的。

除这些设备文件以外，还为一些主要的串行接口卡厂商分配了专用的主设备号。例如，对于 Cyclades 的串行卡，就分配了 19 和 20 两个字符设备主设备号。其中 19 用于 dial in，设备文件名为/dev/ttyC0 至/dev/ttyC31；而 20 则用于“call out”，设备文件名为/dev/cub0 至/dev/cub31。像这样的例子还有很多，在 Linux/Documentation 目录中有一个文件 devices.txt，列出了所有已经分配的设备号，读者可以参考。

上面讲到的这些终端设备都是物理上存在的，即使是“虚拟终端”，最终也要对应到一个物理的终端上。就拿显示器和键盘的组合来说，虽然与传统的终端设备有所不同，但毕竟具备了构成一个终端的物理要素。然而，随着计算机技术的发展，出现了对一种特殊“终端设备”的需求，这种设备在逻辑上是终端设备，可是实际上却不是，所以称为“伪终端”（pseudo tty）。伪终端总是成对地使用的，就好像是一个管道的两端。一端的设备称为“主设备”（master）,其主设备号为 2，设备名为/dev/ptyAX，这里的 A 表示 16 个字母“pqrstuvwxyzPQRST”中的一个，X 则为 16 个 16 进制数字（0～f）之一，这样一共可以有 256 个伪终端主设备。另一端的设备称为“从设备”（slave），其主设备号为 3，设备名则为/dev/ttyAX，同样也是 256 个。每一对伪终端设备，例如/dev/ptyp0 和/dev/ttyp0，就好像是通过一个管道连在一起，其“从设备”一端与普通的终端设备没有什么区别，而“主设备”一端则跟管道文件相似。

那么，为什么要有这样的伪终端设备，又为什么要有主设备和从设备配对呢？让我们考虑当一个 Linux（或 Unix）系统采用 X Window 一类的图形用户界面（GUI）时的情况。在这样的系统里，整个显示屏以及键盘都在一个视窗管理进程的控制之下，显示屏上有若干个用来模拟普通终端的窗口，每个这样的窗口都与一个应用进程例如 shell 相联系。但是每个 shell 进程都以为它的标准输入和标准输出（以及标准出错信息）通道都通向一个终端设备，既不知道也无能力控制显示屏上的窗口。怎么办呢？这就要使用伪终端设备了，图 8.7 是个示意图。

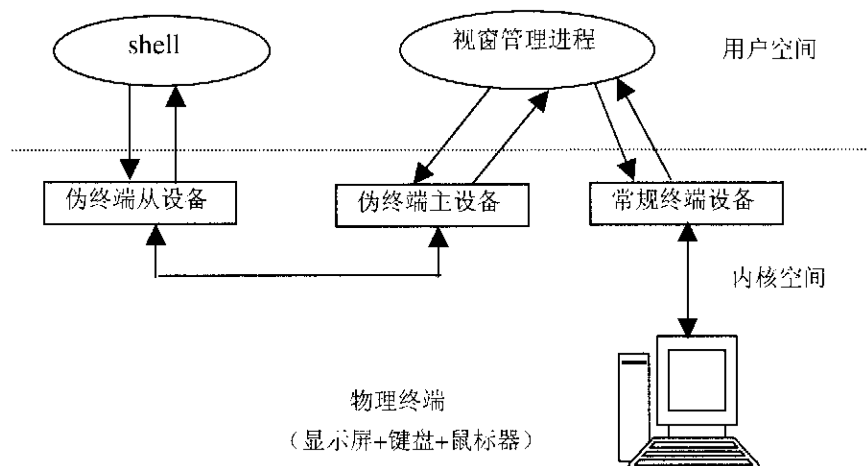


图 8.7 伪终端逻辑示意图

每一对伪终端设备连接着显示屏上的一个窗口和一个应用进程。当视窗管理进程从键盘接收到一个字符时，它先要检查当前的“光标”位置，找出当前“活跃”的窗口和与之对应的伪终端“主设备”（作为一个已打开文件），然后就把从键盘读入的字符写入到伪终端的“主设备”一端。也就是说，对于键盘输入，视窗管理进程起着中转、搬运的作用，写入伪终端“主设备”一端的字符马上就到达了其“从设备”一端。在那里，对于与“从设备”相联系的进程来说，就跟从普通终端设备读入字符一模一样了。反过来，当“从设备”一边的进程有输出时，它的输出通过伪终端的“从设备”到达“主设备”一端，然后由视窗管理进程读取。视窗管理进程从某个伪终端的“主设备”中接收到字符以后，就要根据具体的打开文件号找到显示屏上相应的窗口，换算成显示屏上的位置，再把接收到的字符在这个位置上显示出来。与伪终端“从设备”相联系的进程，根本就不知道它的终端设备到底是一个物理终端，还是实际上只不过是另一个进程。另一方面，伪终端“主设备”一侧的进程也不一定非得把从伪终端“主设备”接收到的内容在显示屏上的某个窗口里显示出来。例如，在网络环境下“从设备”一端的进程可能仍是 shell，而“主设备”一端的进程则可能是 telnetd，它把从伪终端“主设备”接收到的内容“搬运”到一个网络插口中，反之亦然，这个插口也许通过互联网连接到一千公里以外的另一台机器上。

显然，伪终端设备不同于常规的终端设备，它们的驱动程序也理应有所不同，但是实际上却共享同一个 file_operations 数据结构，即 tty_fops。事实上，主设备号为 2、3、4、5（除 7 以外）的字符设备，以及与分配给各家串行口厂商的主设备号相对应的字符设备，全都使用同一个 file_operations 数据结构。这当然并不意味着所有这些设备都使用相同的驱动程序，而是说明：

(1) 终端设备的驱动程序至少可以分成两个或更多个子层，其中最上层是公共的，所以有相同的入口。

(2) 不同类终端设备的读/写操作不同，例如前述由显示器和键盘构成的控制台终端跟外接的笨终端显然不同，所以一定还有一个类似于 file_operations 那样的函数跳转结构。

(3) 终端设备驱动程序也跟信息传输的方式有关，例如通过 RS232 电缆相连的终端跟通过 Modem 相连的终端肯定不一样。所以，一定还存在着另一个与传输方式有关的函数跳转结构。

事实正是这样，除 file_operations 结构以外，每个终端设备还跟另两个数据结构相联系，一个是 tty_driver 数据结构，定义于 include/linux/tty_driver.h:

```
120    struct tty_driver {
121        int magic;                  /* magic number for this structure */
122        const char  *driver_name;
123        const char  *name;
124        int name_base;              /* offset of printed name */
125        short   major;              /* major device number */
126        short   minor_start;        /* start of minor device number*/
127        short   num;                /* number of devices */
128        short   type;               /* type of tty driver */
129        short   subtype;            /* subtype of tty driver */
130        struct termios init_termios; /* Initial termios */
131        int flags;                  /* tty driver flags */
132        int *refcount;              /* for loadable tty drivers */
133        struct proc_dir_entry *proc_entry; /* /proc fs entry */
```

```
134         struct tty_driver *other;   /* only used for the PTY driver */
135
136         /*
137          * Pointer to the tty data structures
138          */
139         struct tty_struct **table;
140         struct termios **termios;
141         struct termios **termios_locked;
142         void *driver_state;         /* only used for the PTY driver */
143
144         /*
145          * Interface routines from the upper tty layer to the tty
146          * driver.
147          */
148         int  (*open)(struct tty_struct * tty, struct file * filp);
149         void (*close)(struct tty_struct * tty, struct file * filp);
150         int  (*write)(struct tty_struct * tty, int from_user,
151                   const unsigned char *buf, int count);
152         void (*put_char)(struct tty_struct *tty, unsigned char ch);
153         void (*flush_chars)(struct tty_struct *tty);
154         int  (*write_room)(struct tty_struct *tty);
155         int  (*chars_in_buffer)(struct tty_struct *tty);
156         int  (*ioctl)(struct tty_struct *tty, struct file * file,
157                 unsigned int cmd, unsigned long arg);
158         void (*set_termios)(struct tty_struct *tty, struct termios * old);
159         void (*throttle)(struct tty_struct * tty);
160         void (*unthrottle)(struct tty_struct * tty);
161         void (*stop)(struct tty_struct *tty);
162         void (*start)(struct tty_struct *tty);
163         void (*hangup)(struct tty_struct *tty);
164         void (*break_ctl)(struct tty_struct *tty, int state);
165         void (*flush_buffer)(struct tty_struct *tty);
166         void (*set_ldisc)(struct tty_struct *tty);
167         void (*wait_until_sent)(struct tty_struct *tty, int timeout);
168         void (*send_xchar)(struct tty_struct *tty, char ch);
169         int (*read_proc)(char *page, char **start, off_t off,
170                   int count, int *eof, void *data);
171         int (*write_proc)(struct file *file, const char *buffer,
172                   unsigned long count, void *data);
173
174         /*
175          * linked list pointers
176          */
177         struct tty_driver *next;
178         struct tty_driver *prev;
179     };
```

可见，结构中给定了该种终端设备的主设备号以及次设备号的范围，并提供了许多函数指针。这

样，对于不同种类的终端设备就有不同的 tty_driver 数据结构，例如控制台终端的 tty_driver 数据结构就是 console_driver。

另一个是 tty_ldisc 数据结构，定义于 include/linux/tty_ldisc.h 中：

```
103	struct tty_ldisc {
104		int	magic;
105		char	*name;
106		int	num;
107		int	flags;
108		/*
109		 * The following routines are called from above.
110		 */
111		int	(*open)(struct tty_struct *);
112		void	(*close)(struct tty_struct *);
113		void	(*flush_buffer)(struct tty_struct *tty);
114		ssize_t	(*chars_in_buffer)(struct tty_struct *tty);
115		ssize_t	(*read)(struct tty_struct * tty, struct file * file,
116				unsigned char * buf, size_t nr);
117		ssize_t	(*write)(struct tty_struct * tty, struct file * file,
118				 const unsigned char * buf, size_t nr);
119		int	(*ioctl)(struct tty_struct * tty, struct file * file,
120				 unsigned int cmd, unsigned long arg);
121		void	(*set_termios)(struct tty_struct *tty, struct termios * old);
122		unsigned int (*poll)(struct tty_struct *, struct file *,
123				     struct poll_table_struct *);
124	
125		/*
126		 * The following routines are called from below.
127		 */
128		void	(*receive_buf)(struct tty_struct *, const unsigned char *cp,
129				       char *fp, int count);
130		int	(*receive_room)(struct tty_struct *);
131		void	(*write_wakeup)(struct tty_struct *);
132	};
```

结构名中的“ldisc”应为“Line Discipline”的缩写，表示“链路规则”的意思。与 file_operations 不同的是，这个结构中不但有供上层调用的函数指针如 open、read、write 等等，还有供下层往上调用的函数指针 receive_buf、receive_room 以及 write_wakeup。此外，结构中还有几个并非函数指针的字段。内核中有个 tty_ldisc 结构数组，用于各种不同的链路规则，包括实际上并不使用链路的“链路规则”，定义于 drivers/char/tty_io.c：

```
119	struct tty_ldisc ldiscs[NR_LDISCS];	/* line disc dispatch table */
```

其大小为 NR_LDISCS，实际上定义为 16，系统在初始化或安装某种驱动模块时通过函数 tty_register_ldisc( )将有关的 tty_ldisc 结构“登记”到这个数组中。

如果把具体终端设备的驱动称为“终端设备层”的话，那么 tty_driver 结构是它与其上层，即常规的设备驱动层之间的界面，而 tty_ldisc 结构则是它与其下层，即物理设备层之间的界面。所谓“链路规则”，实际上就是怎样驱动具体的物理接口。

下面，我们通过打开文件操作来看根据终端的种类转接到具体 tty_ldisc 结构的过程。

如上所述，数据结构 tty_fops 是几乎所有终端设备驱动程序的总枢纽，有着特殊的重要性。这个数据结构是在 drivers/char/tty_io.c 中定义的：

```
407    static struct file_operations tty_fops = {
408        llseek:      tty_lseek,
409        read:        tty_read,
410        write:       tty_write,
411        poll:        tty_poll,
412        ioctl:       tty_ioctl,
413        open:        tty_open,
414        release:     tty_release,
415        fasync:      tty_fasync,
416    };
```

用于打开文件操作的函数是 tty_open( )，因为比较长，我们分段阅读(drivers/char/tty_io.c)。

```
1285    static int tty_open(struct inode * inode, struct file * filp)
1286    {
1287        struct tty_struct *tty;
1288        int noctty, retval;
1289        kdev_t device;
1290        unsigned short saved_flags;
1291        char    buf[64];
1292
1293        saved_flags = filp->f_flags;
1294    retry_open:
1295        noctty = filp->f_flags & O_NOCTTY;
1296        device = inode->i_rdev;
1297        if (device == TTY_DEV) {
1298            if (!current->tty)
1299                return -ENXIO;
1300            device = current->tty->device;
1301            filp->f_flags |= O_NONBLOCK; /* Don't let /dev/tty block */
1302            /* noctty = 1; */
1303        }
1304    #ifdef CONFIG_VT
1305        if (device == CONSOLE_DEV) {
1306            extern int fg_console;
1307            device = MKDEV(TTY_MAJOR, fg_console + 1);
1308            noctty = 1;
1309        }
```

```
1310    #endif
1311        if (device == SYSCONS_DEV) {
1312            struct console *c = console_drivers;
1313            while(c && !c->device)
1314                c = c->next;
1315            if (!c)
1316                            return -ENODEV;
1317                    device = c->device(c);
1318            filp->f_flags |= O_NONBLOCK; /* Don't let /dev/console block */
1319            noctty = 1;
1320        }
1321
1322        if (device == PTMX_DEV) {
1323    #ifdef CONFIG_UNIX98_PTYS
1324
1325            /* find a free pty. */
1326            int major, minor;
1327            struct tty_driver *driver;
1328
1329            /* find a device that is not in use. */
1330            retval = -1;
1331            for ( major = 0 ; major < UNIX98_NR_MAJORS ; major++ ) {
1332                driver = &ptm_driver[major];
1333                for (minor = driver->minor_start ;
1334                     minor < driver->minor_start + driver->num ;
1335                     minor++) {
1336                    device = MKDEV(driver->major, minor);
1337                    if (!init_dev(device, &tty)) goto ptmx_found; /* ok! */
1338                }
1339            }
1340            return -EIO; /* no free ptys */
1341        ptmx_found:
1342            set_bit(TTY_PTY_LOCK, &tty->flags); /* LOCK THE SLAVE */
1343            minor -= driver->minor_start;
1344            devpts_pty_new(driver->other->name_base + minor,
        MKDEV(driver->other->major, minor + driver->other->minor_start));
1345            tty_register_devfs(&pts_driver[major], DEVFS_FL_NO_PERSISTENCE,
1346                           pts_driver[major].minor_start + minor);
1347            noctty = 1;
1348            goto init_dev_done;
1349
1350    #else   /* CONFIG_UNIX_98_PTYS */
1351
1352            return -ENODEV;
1353
1354    #endif  /* CONFIG_UNIX_98_PTYS */
1355        }
1356
```

```
1357        retval = init_dev(device, &tty);
1358        if (retval)
1359            return retval;
1360
```

找到代表着目标设备的文件节点以后，其 inode 结构中的 i_rdev 字段就是目标设备的设备号。首先要看这是不是当前进程的控制终端，常数 TTY_DEV 定义于 drivers/char/tty_io.c：

```
108     #define TTY_DEV MKDEV(TTYAUX_MAJOR,0)
```

就是说，主设备号为 5，次设备号为 0，这就是/dev/tty，表示当前进程的“控制终端”，而并不指向任何物理意义上的终端。一个进程通常都有个“控制终端”。如果没有，那么打开的第一个终端设备一般就成为其控制终端，但是那得要使用表示具体终端的设备文件名才行。有时候虽然一个进程尚无控制终端，并且又要打开一个终端设备，可是却并不要将它作为控制终端，在这种情况下就可以在系统调用 open( )的参数 flags 中将一个标志 O_NOCTTY 设成 1，这里的 1295 行将这个标志位的值保存在一个局部量 noctty 中备用。另一方面，每个进程的 task_structure 数据结构中有个指针 tty，指向代表着其当前控制终端的 tty_struct 数据结构。如果这个指针为 0，那就说明这个进程还没有控制终端，所以返回出错代码-ENXIO，表示没有这么个设备。如果要打开的确实是当前进程的“控制终端”，那就已经不是第一次打开了，这里把通过/dev/tty 建立的连接强制设成 O_NONBLOCK。

编译选择项 CONFIG_VT 表示系统中是否配备了前述由显示器和键盘构成的虚拟终端。常数 CONSOLE_DEV 和 SYSCON_DEV 的定义为(见 drivers/char/n_tty.c)：

```
48      #define CONSOLE_DEV MKDEV(TTY_MAJOR,0)
49      #define SYSCONS_DEV  MKDEV(TTYAUX_MAJOR,1)
```

就是说，如果主设备号为 4 而次设备号为 0，即/dev/tty0（表示当前虚拟控制台），则将其替换成具体的设备号。在支持虚拟控制台的内核中有个全局量 fg_console，表示当前的“前台控制台”。这个变量的数值是从 0 开始的，而各个具体虚拟控制台的次设备号却从 1 开始，所以二者在数值上差 1。这样，1307 行就根据 fg_console 的数值还原了当前虚拟控制台的设备号。另一方面，由于这个虚拟控制台原来就已打开，所以把 noctty 设成 1，保持其原状不变。主设备号为 5 而次设备号为 1 即/dev/console，用于外接的控制台。我们在前面曾经提到过，/dev/console 一般是连接到/dev/tty0 的。但是，在不同的版本里情况有所不同。如果/dev/console 不是连接到/dev/tty0，而是主设备号为 5，次设备号为 1，那么内核在初始化过程中安装模块时会通过一个函数 register_console( )登记用作控制台的终端设备，把一个 console 数据结构挂到内核中的 console_drivers 队列中，此后内核便会将需要显示的出错信息（通过 printk( )显示的信息）输出到所登记的设备上。我们在后面要讲到 console 数据结构，但是在这里只要知道这个数据结构里有个字段 device，就是具体终端设备的设备号就行了。所以，如果要打开的终端设备是这样的/dev/console，就要在 console_drivers 队列中找到第一个设备号不会 0 的 console 结构，它所提供的设备号就是当前系统控制台的设备号。

最后，常数 PTMX_DEV 的定义见 drivers/char/tty_io.c：

```
110     #define PTMX_DEV MKDEV(TTYAUX_MAJOR,2)
```

在 Documents/device.txt 文件中可以查到，主设备号为 5 而次设备号为 2 的字符设备文件是/dev/ptmx，是用于伪终端主设备的。我们在前面已经介绍过伪终端设备，其主设备和从设备的主设备号分别为 2 和 3，并且因此而可以有 256 对伪终端设备（次设备号为 8 位）。这实际上是 BSD 版本的做法，现在也一直在沿用，但是 system V 采用了一种不同的方法。在 system V 中设置了一个总的伪终端主设备文件/dev/ptmx，但是它的作用与上述/dev/console 相似，只是提供了一个总的入口，而实际的设备号要在打开文件时临时分配，这样就可以有更多的设备号用于伪终端设备。在早期的应用中，有上百对的伪终端设备已经够用了，所以 system V 这种方法的优点并不明显，但是，随着网络技术和应用的发展，后者的优越性却显得突出起来。用 Unix/Linux 系统作为网络服务器时，可能会要求数百个甚至上千个连接，这时候 256 个伪终端主设备号就不够用了，而采用 system V 的方法就不受这个限制。当然，采用 BSD 的方法也可以通过增加用于伪终端设备的主设备号个数来扩充它的容量，但是那样在应用程序中就要考虑对大量伪终端设备文件的调度和管理。例如，你怎么知道/dev/ptyr32 已经在使用中，而/dev/ptyx87 还空闲着呢？而 systom V 的方法则实际上把这部分工作移到了内核中，效率当然要高一些，并且应用程序也可以简化。所以，在新的 Unix 版本，即 Unix98 中采用了 System V 的方案并加以配套完善，并且又将主设备号 125～135 分配用于伪终端主设备，136～143 用于次设备，这样理论上的容量就可以达到 8×256，即 2048 对伪终端设备。对于 Linux 内核，这是作为一个可选项提供的，相应的条件编译选择项就是 CONFIG_UNIX98_PTYS，不过这需要 2.1 版或以上的 C 程序库 glibc 配套（glibc 为用户程序提供若干库函数，例如 ptsname( )返回与一个已经打开的伪终端主设备相对应的次设备文件名）。如果选择了这个可选项，那么内核在初始化时根据实际用于伪终端主设备号的个数（由常数 UNIX98_NR_MAJORS 决定）初始化两个 tty_driver 结构(见下)数组 ptm_driver[ ]和 pts_driver[ ]，分别用于主设备和从设备，然后在用户进程通过/dev/ptmx 打开一个伪终端主设备时再动态加以分配。代码中第 1331 行开始的 for 循环扫描 ptm_driver[ ]中的每个元素，即每个主设备号，并且对于每个主设备号依次尝试各个次设备号，如果以某个主设备号和次设备号的组合调用 init_dev( )成功，则一个新的伪终端主设备就打开成功了。但是，接着还要转到 ptmx_found 处为进一步打开次设备作好准备。数组 ptm_driver[ ]和 pts_driver[ ]中的每个元素都是一个 tty_driver 数据结构，结构中有个指针 other 用来指向与之配对的另一方，所以 1344 行的 driver 指向 ptm_driver[ ]中的某个元素，而 driver->other 则指向 pts_driver[ ]中与之配对的元素。此外，函数 devpts_pty_new( )为相应的次设备在内存中创建一个 inode 数据结构，tty_reqister_devfs 则为之创建一个 devfs 设备节点，例如/dev/pts/0，/dev/pts/134，等等。注意，终端的号码是连续编号的。由于篇幅所限，我们在这里就不深入到这两个函数中去了，有需要或有兴趣的读者可以自行阅读。

除对/dev/ptmx 作特殊处理外，对其他终端设备（包括伪终端设备）的打开文件操作在取得了最终的设备号以后都在 1357 行调用 init_dev( )，为需要打开的终端设备建立一个(或找到其) tty_struct 数据结构。

每个已打开的终端设备都由一个 tty_struct 数据结构代表，这种数据结构定义于 include/linux/tty.h：

```
245     /*
246      * Where all of the state associated with a tty is kept while the tty
247      * is open.  Since the termios state should be kept even if the tty
```

```
248      * has been closed --- for things like the baud rate, etc --- it is
249      * not stored here, but rather a pointer to the real state is stored
250      * here.  Possible the winsize structure should have the same
251      * treatment, but (1) the default 80x24 is usually right and (2) it's
252      * most often used by a windowing system, which will set the correct
253      * size each time the window is created or resized anyway.
254      * IMPORTANT: since this structure is dynamically allocated, it must
255      * be no larger than 4096 bytes.  Changing TTY_FLIPBUF_SIZE will change
256      * the size of this structure, and it needs to be done with care.
257      *                       - TYT, 9/14/92
258      */
259     struct tty_struct {
260         int magic;
261         struct tty_driver driver;
262         struct tty_ldisc ldisc;
263         struct termios *termios, *termios_locked;
264         int pgrp;
265         int session;
266         kdev_t  device;
267         unsigned long flags;
268         int count;
269         struct winsize winsize;
270         unsigned char stopped:1, hw_stopped:1, flow_stopped:1, packet:1;
271         unsigned char low_latency:1, warned:1;
272         unsigned char ctrl_status;
273 
274         struct tty_struct *link;
275         struct fasync_struct *fasync;
276         struct tty_flip_buffer flip;
277         int max_flip_cnt;
278         int alt_speed;      /* For magic substitution of 38400 bps */
279         wait_queue_head_t write_wait;
280         wait_queue_head_t read_wait;
281         struct tq_struct tq_hangup;
282         void *disc_data;
283         void *driver_data;
284         struct list_head tty_files;
285 
286     #define N_TTY_BUF_SIZE 4096
287 
288         /*
289          * The following is data for the N_TTY line discipline.  For
290          * historical reasons, this is included in the tty structure.
291          */
292         unsigned int column;
293         unsigned char lnext:1, erasing:1, raw:1, real_raw:1, icanon:1;
294         unsigned char closing:1;
295         unsigned short minimum_to_wake;
```

```
296         unsigned overrun_time;
297         int num_overrun;
298         unsigned long process_char_map[256/(8*sizeof(unsigned long))];
299         char *read_buf;
300         int read_head;
301         int read_tail;
302         int read_cnt;
303         unsigned long read_flags[N_TTY_BUF_SIZE/(8*sizeof(unsigned long))];
304         int canon_data;
305         unsigned long canon_head;
306         unsigned int canon_column;
307         struct semaphore atomic_read;
308         struct semaphore atomic_write;
309         spinlock_t read_lock;
310     };
```

这里先介绍一下其中几个特别重要的成分，结构中其他字段的作用随着代码的阅读自会变得清楚起来。首先是一个 tty_driver 数据结构(261 行)。如前所述，每种终端设备都有自己的 tty_driver 结构，结构中主要是一些函数指针，确定了对具体终端类型的一整套操作。内核中有个链表 tty_drivers，系统在初始化时，或安装某种终端设备的驱动模块时，通过函数 tty_register_driver( )将各种终端设备的 tty_driver 结构登记到这个链表中。每当新打开一个终端设备时，就要根据其设备号通过函数 get_tty_driver( )在这个链表中找到相应的 tty_driver 结构，并把它复制到具体的 tty_struct 结构中。第二个重要成分就是 tty_ldisc 数据结构（262 行）。如前所述，内核中有个 tty_ldisc 结构数组 ldiscs[ ]，当新创建一个 tty_struct 结构时，就从该数组中把相应的 tty_ldisc 结构复制到 tty_struct 结构中的这个成分中。第三个重要成分是指针 termios，指向一个 termios 结构。每个逻辑意义上的终端设备接口，如串行口、图形卡和键盘的组合乃至伪终端设备的两端，都有一个 termios 数据结构。这个数据结构在某种程度上可以看作是对 tty_ldisc 结构的补充，它规定了对接口上输入和输出的每个字符所作的处理以及传输的速度，即“波特率”。数据结构的定义在 include/asm-386/termbits.h 中：

```
10      #define NCCS 19
11      struct termios {
12          tcflag_t c_iflag;       /* input mode flags */
13          tcflag_t c_oflag;       /* output mode flags */
14          tcflag_t c_cflag;       /* control mode flags */
15          tcflag_t c_lflag;       /* local mode flags */
16          cc_t c_line;            /* line discipline */
17          cc_t c_cc[NCCS];        /* control characters */
18      };
```

同一文件中定义了用于这个结构中各个字段的许多常数（大多是标志位），例如用于 c_lflag 字段的标志位 ISG 为 0000001，ICANON 为 0000002，ECHO 为 0000010。如果将 c_lflag 字符中的这些标志位清成 0，则相应终端设备就工作于所谓“原始模式”(raw mode)，否则就工作于所谓“加工模式”(cooked mode)。有关详情在阅读有关代码时还会讲到。限于篇幅，就不在这里列出这些常数的定义了。在 tty_struct 结构中还有个重要的成分 flip，是个 tty_flip_buffer 数据结构。它是终端设备的输入缓冲区，

底层的中断服务程序将接收到的字符存储于这个缓冲区中，供其上层读取。这就是我们以前提到过的上层的同步读操作与底层的异步读操作之间的交汇点，其定义在 include/linux/tty.h 中：

```
132	/*
133	 * This is the flip buffer used for the tty driver.  The buffer is
134	 * located in the tty structure, and is used as a high speed interface
135	 * between the tty driver and the tty line discipline.
136	 */
137	#define TTY_FLIPBUF_SIZE 512
138
139	struct tty_flip_buffer {
140		struct tq_struct	tqueue;
141		struct semaphore	pty_sem;
142		char		*char_buf_ptr;
143		unsigned char	*flag_buf_ptr;
144		int		count;
145		int		buf_num;
146		unsigned char	char_buf[2*TTY_FLIPBUF_SIZE];
147		char		flag_buf[2*TTY_FLIPBUF_SIZE];
148		unsigned char	slop[4]; /* N.B. bug overwrites buffer by 1 */
149	};
```

与 flip 相应的另一个成分是字符位图 process_char_map，这个位图中的每一位都对应着一个 8 位字符，所以位图的大小为 32 字节。如果某个字符在这个位图中的对应位为 1，就表示可能要对这个字符作出一些特殊的处理或反应（往往只是在“加工模式”下才起作用）。

随着代码的阅读，数据结构中其他字段的作用会慢慢变得清楚起来。函数 init_dev( )的代码在 drivers/char/tty_io.c 中，这也是个比较长的函数，也得要分段阅读。

[tty_open( ) > init_dev( )]

```
806	static int init_dev(kdev_t device, struct tty_struct **ret_tty)
807	{
808		struct tty_struct *tty, *o_tty;
809		struct termios *tp, **tp_loc, *o_tp, **o_tp_loc;
810		struct termios *ltp, **ltp_loc, *o_ltp, **o_ltp_loc;
811		struct tty_driver *driver;
812		int retval=0;
813		int idx;
814
815		driver = get_tty_driver(device);
816		if (!driver)
817			return -ENODEV;
818
819		idx = MINOR(device) - driver->minor_start;
820
821		/*
```

```
822          * Check whether we need to acquire the tty semaphore to avoid
823          * race conditions.  For now, play it safe.
824          */
825         down_tty_sem(idx);
826
827         /* check whether we're reopening an existing tty */
828         tty = driver->table[idx];
829         if (tty) goto fast_track;
830
831         /*
832          * First time open is complex, especially for PTY devices.
833          * This code guarantees that either everything succeeds and the
834          * TTY is ready for operation, or else the table slots are vacated
835          * and the allocated memory released.  (Except that the termios
836          * and locked termios may be retained.)
837          */
838
839         o_tty = NULL;
840         tp = o_tp = NULL;
841         ltp = o_ltp = NULL;
842
843         tty = alloc_tty_struct( );
844         if(!tty)
845             goto fail_no_mem;
846         initialize_tty_struct(tty);
847         tty->device = device;
848         tty->driver = *driver;
849
850         tp_loc = &driver->termios[idx];
851         if (!*tp_loc) {
852             tp = (struct termios *) kmalloc(sizeof(struct termios),
853                             GFP_KERNEL);
854             if (!tp)
855                 goto free_mem_out;
856             *tp = driver->init_termios;
857         }
858
859         ltp_loc = &driver->termios_locked[idx];
860         if (!*ltp_loc) {
861             ltp = (struct termios *) kmalloc(sizeof(struct termios),
862                             GFP_KERNEL);
863             if (!ltp)
864                 goto free_mem_out;
865             memset(ltp, 0, sizeof(struct termios));
866         }
867
```

调用参数有两个。一个是设备号，由于所有的终端设备都共用同一个 file_operations 结构，从而

同一个函数 tty_open( )，这里需要进一步根据设备号来区分具体的设备和操作。另一个参数是双重指针，指向一个 tty_struct 指针，目的是用来返回一个 tty_struct 数据结构。

简言之，init_dev( )的任务就是根据设备号找到或创建目标终端设备的 tty_struct 数据结构，并且执行在打开该设备时所需的附加操作。首先是通过 get_tty_driver( )根据设备号从 tty_drivers 链表中找到相应的 tty_driver 数据结构(815 行)，这个结构中有个指针 table，指向一个 tty_struct 结构指针数组。数组中包含了所有该种终端设备的 tty_struct 结构指针。举例来说，主设备号为 5 的"辅助 TTY 设备"实际上包括了好几种不同的终端设备，其中次设备号 64～255 代表着 192 个"呼出型"串行口终端设备，设备文件为 /dev/cua0 至/dev/cua191。所以，在链表 tty_drivers 中就有一个代表着呼出型终端设备的 tty_driver 数据结构，而这个结构中的指针就指向一个大小为 192 的 tty_struct 结构指针数组。由于这种终端设备的次设备号不是从 0 开始的，所以要有一个 minor_start 字段，用以说明其起始次设备号，而具体的次设备号与起始次设备的差就用作访问该数组的下标（见 819 行）。如果一个终端设备已经打开，即其 tty_struct 结构已经存在，那么数组中相应的指针就不为 0，那就不需要创建了(见 828 和 829 行)，否则就要通过 alloc_tty_struct( )分配空间，然后就是对这个数据结构的初始化。注意 848 行是数据结构的赋值，是把整个 tty_driver 数据结构复制到目标设备的 tty_struct 结构的内部。这样一来，前述两个函数跳转结构之一就与目标设备挂上了钩。函数 initialize_tty_struct( )的代码也在 tty_io.c 中：

[tty_open( ) > init_dev( ) > initialize_tty_struct( )]

```
1958    static void initialize_tty_struct(struct tty_struct *tty)
1959    {
1960        memset(tty, 0, sizeof(struct tty_struct));
1961        tty->magic = TTY_MAGIC;
1962        tty->ldisc = ldiscs[N_TTY];
1963        tty->pgrp = -1;
1964        tty->flip.char_buf_ptr = tty->flip.char_buf;
1965        tty->flip.flag_buf_ptr = tty->flip.flag_buf;
1966        tty->flip.tqueue.routine = flush_to_ldisc;
1967        tty->flip.tqueue.data = tty;
1968        init_MUTEX(&tty->flip.pty_sem);
1969        init_waitqueue_head(&tty->write_wait);
1970        init_waitqueue_head(&tty->read_wait);
1971        tty->tq_hangup.routine = do_tty_hangup;
1972        tty->tq_hangup.data = tty;
1973        sema_init(&tty->atomic_read, 1);
1974        sema_init(&tty->atomic_write, 1);
1975        spin_lock_init(&tty->read_lock);
1976        INIT_LIST_HEAD(&tty->tty_files);
1977    }
```

这里 1962 行又是数据结构的赋值，这一次是把整个 tty_ldisc 数据结构复制到了目标设备的 tty_struct 结构的内部。这样，前述的两个函数跳转结构就都与目标设备挂上了钩。不管是什么种类的终端设备，其默认的 tty_ldisc 数据结构都是 ldiscs[N_TTY]，这里 N_TTY 定义为 0。这就是虚拟终端，

即 VGA 卡加键盘所采用的链路规则，这种“链路”的规则就是不存在物理的链路。至于其他终端设备，则可以在以后通过系统调用 ioctl( )设置采用特殊的链路规则。

回到 init_dev( )的代码中。如前所述，在 tty_driver 结构中有个指针 termios 指向一个 termios 指针数组，这个数组也是以终端号（次设备号减去该类终端设备的起始次设备号）为下标的。与此相平行，还有个指针 termios_locked 指向另一个 termios 结构指针数组。如果这两个数组中对应于正欲打开的终端设备的 termios 结构指针为 0，就表示尚未为之创设 termios 数据结构，所以要为之分配空间（见 852 行和 861 行）并初始化。其中属于 termios[ ]的数据结构从 driver 结构中的一个“样板”termios 结构 init_termios 复制（见 856 行），而属于 termios_locked[ ]的数据结构则初始化成全 0（见 865 行）。

再往下看(tty_io.c)：

[tty_open( ) > init_dev( )]

```
868         if (driver->type == TTY_DRIVER_TYPE_PTY) {
869             o_tty = alloc_tty_struct( );
870             if (!o_tty)
871                 goto free_mem_out;
872             initialize_tty_struct(o_tty);
873             o_tty->device = (kdev_t) MKDEV(driver->other->major,
874                          driver->other->minor_start + idx);
875             o_tty->driver = *driver->other;
876 
877             o_tp_loc  = &driver->other->termios[idx];
878             if (!*o_tp_loc) {
879                 o_tp = (struct termios *)
880                     kmalloc(sizeof(struct termios), GFP_KERNEL);
881                 if (!o_tp)
882                     goto free_mem_out;
883                 *o_tp = driver->other->init_termios;
884             }
885 
886             o_ltp_loc = &driver->other->termios_locked[idx];
887             if (!*o_ltp_loc) {
888                 o_ltp = (struct termios *)
889                     kmalloc(sizeof(struct termios), GFP_KERNEL);
890                 if (!o_ltp)
891                     goto free_mem_out;
892                 memset(o_ltp, 0, sizeof(struct termios));
893             }
894 
895             /*
896              * Everything allocated ... set up the o_tty structure.
897              */
898             driver->other->table[idx] = o_tty;
899             if (!*o_tp_loc)
900                 *o_tp_loc = o_tp;
901             if (!*o_ltp_loc)
```

```
902                 *o_ltp_loc = o_ltp;
903             o_tty->termios = *o_tp_loc;
904             o_tty->termios_locked = *o_ltp_loc;
905             (*driver->other->refcount)++;
906             if (driver->subtype == PTY_TYPE_MASTER)
907                 o_tty->count++;
908
909             /* Establish the links in both directions */
910             tty->link   = o_tty;
911             o_tty->link = tty;
912         }
913
```

对于打开文件操作，伪终端是有其特殊性的，所以如果要打开的是伪终端设备（不论是主设备或次设备）就要进行一些特殊的处理。打开伪终端设备时，tty_struct 结构是成对地分配、创建的，这样才能使两个数据结构预先“背靠背”地互相挂上钩。以打开一个伪终端主设备为例，在创建了主设备一侧的 tty_struct 结构 tty(见前面的 843 行)以后，还要创建从设备一侧的 o_tty（见 869 行）。这里 o_tty 显然是表示“the other tty”(另一个 tty)的意思。伪终端主设备和从设备的 tty_driver 数据结构都通过指针 other 指向对方，所以如果 875 行中的 driver 指向主设备的 tty_driver 结构，则 driver->other 指向与其配对的从设备的 tty_driver 结构。与此相似，在 tty_struct 结构中也有个指针 link，可以用来互相指向对方建立起伪终端主/从设备间的连接(见 910～911 行)。这样，在打开主设备的同时就分配好了从设备的 tty_struct 结构，下一次要打开从设备的时候就会在前面的 828 行发现所需的 tty_struct 结构已经存在，从而跳过创建 tty_struct 结构等操作，转到 fast_track 处(在 956 行)。

我们继续往下看(tty_io.c)：

[tty_open( ) > init_dev( )]

```
914         /*
915          * All structures have been allocated, so now we install them.
916          * Failures after this point use release_mem to clean up, so
917          * there's no need to null out the local pointers.
918          */
919         driver->table[idx] = tty;
920
921         if (!*tp_loc)
922             *tp_loc = tp;
923         if (!*ltp_loc)
924             *ltp_loc = ltp;
925         tty->termios = *tp_loc;
926         tty->termios_locked = *ltp_loc;
927         (*driver->refcount)++;
928         tty->count++;
929
930         /*
931          * Structures all installed ... call the ldisc open routines.
```

```
932          * If we fail here just call release_mem to clean up.  No need
933          * to decrement the use counts, as release_mem doesn't care.
934          */
935         if (tty->ldisc.open) {
936             retval = (tty->ldisc.open)(tty);
937             if (retval)
938                 goto release_mem_out;
939         }
940         if (o_tty && o_tty->ldisc.open) {
941             retval = (o_tty->ldisc.open)(o_tty);
942             if (retval) {
943                 if (tty->ldisc.close)
944                     (tty->ldisc.close)(tty);
945                 goto release_mem_out;
946             }
947         }
948         goto success;
949 
950         /*
951          * This fast open can be used if the tty is already open.
952          * No memory is allocated, and the only failures are from
953          * attempting to open a closing tty or attempting multiple
954          * opens on a pty master.
955          */
956     fast_track:
957         if (test_bit(TTY_CLOSING, &tty->flags)) {
958             retval = -EIO;
959             goto end_init;
960         }
961         if (driver->type == TTY_DRIVER_TYPE_PTY &&
962             driver->subtype == PTY_TYPE_MASTER) {
963             /*
964              * special case for PTY masters: only one open permitted,
965              * and the slave side open count is incremented as well.
966              */
967             if (tty->count) {
968                 retval = -EIO;
969                 goto end_init;
970             }
971             tty->link->count++;
972         }
973         tty->count++;
974         tty->driver = *driver; /* N.B. why do this every time?? */
975 
976     success:
977         *ret_tty = tty;
978 
979         /* All paths come through here to release the semaphore */
```

```
980     end_init:
981         up_tty_sem(idx);
982         return retval;
983
984         /* Release locally allocated memory ... nothing placed in slots */
985     free_mem_out:
986         if (o_tp)
987             kfree(o_tp);
988         if (o_tty)
989             free_tty_struct(o_tty);
990         if (ltp)
991             kfree(ltp);
992         if (tp)
993             kfree(tp);
994         free_tty_struct(tty);
995
996     fail_no_mem:
997         retval = -ENOMEM;
998         goto end_init;
999
1000        /* call the tty release_mem routine to clean out this slot */
1001    release_mem_out:
1002        printk("init_dev: ldisc open failed, clearing slot %d\n", idx);
1003        release_mem(tty, idx);
1004        goto end_init;
1005    }
```

这里将新创建的 tty_struct 结构“安装”在其所属 tty_driver 结构的 table[ ]数组中相应的位置上(919 行)，将新创建的 termios 结构安装在相应的数组中以及 tty_struct 结构中（921 行和 925 行）。最终完成对 tty_struct 结构的初始化以后，可能还要执行下层的打开文件操作。如果 tty_ldisc 数据结构中的函数指针 open 非 0，就要通过这个函数进行链路的初始化。如前所述，不管是哪一种终端设备，开始时总是采用与下标 N_TTY 对应的 tty_ldisc 结构。实际上，这个结构是 tty_ldisc_N_TTY，其中的指针 open 指向 n_tty_open( )，这个函数的代码在 drivers/char/n_tty.c 中：

[tty_open( ) > init_dev( ) > n_tty_open( )]

```
860     static int n_tty_open(struct tty_struct *tty)
861     {
862         if (!tty)
863             return -EINVAL;
864
865         if (!tty->read_buf) {
866             tty->read_buf = alloc_buf( );
867             if (!tty->read_buf)
868                 return -ENOMEM;
869         }
```

```
870         memset(tty->read_buf, 0, N_TTY_BUF_SIZE);
871         reset_buffer_flags(tty);
872         tty->column = 0;
873         n_tty_set_termios(tty, 0);
874         tty->minimum_to_wake = 1;
875         tty->closing = 0;
876         return 0;
877     }
```

读者在前面看到，tty_driver 结构中有个成分 flip，是个 tty_flip_buffer 数据结构，它就是终端设备的输入缓冲区。终端设备的输入有两种方式，一种叫“原始模式”，另一种叫“加工模式”，在实际应用中大多工作于加工模式。Unix 的作者给“加工模式”取了个很形象的名字，叫“cooked”即“经过烹调的”。它把从终端设备键盘接收到的字符流比喻作"生米"，而把经过处理以后的字符流比喻成“熟饭”。如果把 tty_driver 结构中的 tty_flip_buffer 比喻作煮饭的锅，那么还得有个盘子来存放煮熟了的饭。为了这个目的，在 tty_driver 结构中设置了一个指针 read_buf，让它指向一个缓冲区，这就是盛饭的盘子。在打开终端设备时，如果发现这个缓冲区尚未分配就要为之分配一个(866 行)，缓冲区的大小为一个页面。与这个缓冲页面相对应，在 tty_struct 结构中还有个位图 read_flags[ ]，位图中的每一位对应着上述缓冲区中的每个字符，在“规范模式”下用来分隔不同的缓冲行。此外，还有一些配合缓冲区使用的字段，例如 read_head 和 canon_read，都是缓冲区中当前读出位置的下标，等等。所有这些字段都要加以初始化，这是由 reset_buffer_flags( )完成的(见 n_tty.c)：

[tty_open( ) > init_dev( ) > n_tty_open( ) > reset_buffer_flags( )]

```
118     /*
119      * Reset the read buffer counters, clear the flags,
120      * and make sure the driver is unthrottled. Called
121      * from n_tty_open( ) and n_tty_flush_buffer( ).
122      */
123     static void reset_buffer_flags(struct tty_struct *tty)
124     {
125         unsigned long flags;
126 
127         spin_lock_irqsave(&tty->read_lock, flags);
128         tty->read_head = tty->read_tail = tty->read_cnt = 0;
129         spin_unlock_irqrestore(&tty->read_lock, flags);
130         tty->canon_head = tty->canon_data = tty->erasing = 0;
131         memset(&tty->read_flags, 0, sizeof tty->read_flags);
132         check_unthrottle(tty);
133     }
```

所涉及字段的作用在阅读 tty_read( )的代码时就会清楚，现在只是初始化。

在 n_tty_open( )中还有个函数要执行，那就是 n_tty_set_termios( )。它的作用主要是根据终端设备的 termios 数据结构设置其 tty_struct 结构中的字符位图 process_char_map 和其他几个标志位(而不是“设置 termios 结构”)。位图 process_char_map 的大小是 32 字节，共 256 位，其中的每一位都对应着一个

字符。如果位图中的某一位为 1，便说明与其对应的字符在“烹调”中需要加以特殊的处理。这个函数的代码也在 drivers/char/n_tty.c 中：

[tty_open( ) > init_dev( ) > n_tty_open( ) > n_tty_set_termios( )]

```
786    static void n_tty_set_termios(struct tty_struct *tty, struct termios * old)
787    {
788        if (!tty)
789            return;
790
791        tty->icanon = (L_ICANON(tty) != 0);
792        if (test_bit(TTY_HW_COOK_IN, &tty->flags)) {
793            tty->raw = 1;
794            tty->real_raw = 1;
795            return;
796        }
797        if (I_ISTRIP(tty) || I_IUCLC(tty) || I_IGNCR(tty) ||
798            I_ICRNL(tty) || I_INLCR(tty) || L_ICANON(tty) ||
799            I_IXON(tty) || L_ISIG(tty) || L_ECHO(tty) ||
800            I_PARMRK(tty)) {
801            cli( );
802            memset(tty->process_char_map, 0, 256/8);
803
804            if (I_IGNCR(tty) || I_ICRNL(tty))
805                set_bit('\r', &tty->process_char_map);
806            if (I_INLCR(tty))
807                set_bit('\n', &tty->process_char_map);
808
809            if (L_ICANON(tty)) {
810                set_bit(ERASE_CHAR(tty), &tty->process_char_map);
811                set_bit(KILL_CHAR(tty), &tty->process_char_map);
812                set_bit(EOF_CHAR(tty), &tty->process_char_map);
813                set_bit('\n', &tty->process_char_map);
814                set_bit(EOL_CHAR(tty), &tty->process_char_map);
815                if (L_IEXTEN(tty)) {
816                    set_bit(WERASE_CHAR(tty),
817                        &tty->process_char_map);
818                    set_bit(LNEXT_CHAR(tty),
819                        &tty->process_char_map);
820                    set_bit(EOL2_CHAR(tty),
821                        &tty->process_char_map);
822                    if (L_ECHO(tty))
823                        set_bit(REPRINT_CHAR(tty),
824                            &tty->process_char_map);
825                }
826            }
827            if (I_IXON(tty)) {
```

```
828                 set_bit(START_CHAR(tty), &tty->process_char_map);
829                 set_bit(STOP_CHAR(tty), &tty->process_char_map);
830             }
831             if (L_ISIG(tty)) {
832                 set_bit(INTR_CHAR(tty), &tty->process_char_map);
833                 set_bit(QUIT_CHAR(tty), &tty->process_char_map);
834                 set_bit(SUSP_CHAR(tty), &tty->process_char_map);
835             }
836             clear_bit(__DISABLED_CHAR, &tty->process_char_map);
837             sti();
838             tty->raw = 0;
839             tty->real_raw = 0;
840         } else {
841             tty->raw = 1;
842             if ((I_IGNBRK(tty) || (!I_BRKINT(tty) && !I_PARMRK(tty))) &&
843                 (I_IGNPAR(tty) || !I_INPCK(tty)) &&
844                 (tty->driver.flags & TTY_DRIVER_REAL_RAW))
845                 tty->real_raw = 1;
846             else
847                 tty->real_raw = 0;
848         }
849     }
```

这个函数的代码中使用了许多宏定义，这些宏定义以后也常要用到。这些定义都在 include/linux/tty.h 和 include/asm-i386/termbits.h 中，虽然数量不少，却很有规则，先看几个基本操作：

```
183     #define _I_FLAG(tty,f)    ((tty)->termios->c_iflag & (f))
184     #define _O_FLAG(tty,f)    ((tty)->termios->c_oflag & (f))
185     #define _C_FLAG(tty,f)    ((tty)->termios->c_cflag & (f))
186     #define _L_FLAG(tty,f)    ((tty)->termios->c_lflag & (f))
```

这些宏操作分别检验 termios 结构中 c_iflag(输入)、c_oflag(输出)、c_cflag(控制)以及 c_lflag（本地）字段中的某一位。在这个基础上，例如，L_ICANON(tty)的定义为：

```
230     #define L_ICANON(tty)     _L_FLAG((tty),ICANON)
```

就是说，前缀 L_表示要检查的标志位在 c_lflag 中，而标志位为 ICANON。由此类推，I_IGNCR 表示 c_iflag 中的 IGNCR 标志位，L_ECHO 表示 c_lflag 中的 ECHO 标志位，等等。这些标志位的意义和作用大致如下：

L_ICANON　表示“canonical mode”或“规范模式”，这就是“加工模式”，对输入的字符要加以“烹调”。

I_ISTRIP　表示强制将输入字符的最高位设成 0，使它变成 7 位编码。

I_IUCLC　表示将接收到的大字字母转换成小写。

I_IGNCR　表示将接收到的“回车”字符“\”丢弃。

I_ICRNL　表示将接收到的"\r"字符转换成"\n"（如果 I_IGNCR 为 0 的话）。
I_INLCR　表示将接收到的"\n"字符转换成"\r"。
I_IXON　表示如果接收到 CTRL_S 字符就暂停输出，直到接收到 CTRL_Q 时再恢复输出。
L_ISIG　表示如果接收到 CTRL_C 等控制字符就向在该终端设备上运行的进程发出信号。
L_ECHO　表示"本地回送"，就是在从终端设备接收到一个字符时就立即将该字符回送到该终端设备上。

此外，在不同的系统中或者不同的键盘上使用的控制字符有可能不同。例如，在屏幕上显示大量信息时，一般可以按 Ctrl-S 暂停屏幕上显示的信息向上滚动，然后可以按 Ctrl-Q 继续输出。可是，在有些键盘上也许不是 Ctrl-S 和 Ctrl-Q，而是别的什么。所以在 termios 结构中还有个数组 c_cc[ ]，用来定义各种控制字符的代码。下面是取自 include/linux/tty.h 和 include/asm-i386/termbits.h 的两个片段，可以用来说明这个数组的用途。

```
173     #define START_CHAR(tty) ((tty)->termios->c_cc[VSTART])
174     #define STOP_CHAR(tty) ((tty)->termios->c_cc[VSTOP])

 29     #define VSTART 8
 30     #define VSTOP 9
```

这表示在目标终端设备上用于这两种目的的控制字符分别是相应 termios 结构中的 c_cc[8]和 c_cc[9]。不过，这个数组不完全是用于控制字符，其中也有几个元素用于其他控制目的。读者在下一节中将会看到这些标志位和控制字符在"烹调"中的作用。

最后，init_dev( )通过参数 ret_tty 返回指向 tty_struct 结构的指针。我们回到 tty_open( )的代码中继续往下看(tty_io.c)：

[tty_open( )]

```
1361    #ifdef CONFIG_UNIX98_PTYS
1362    init_dev_done:
1363    #endif
1364        filp->private_data = tty;
1365        file_move(filp, &tty->tty_files);
1366        check_tty_count(tty, "tty_open");
1367        if (tty->driver.type == TTY_DRIVER_TYPE_PTY &&
1368            tty->driver.subtype == PTY_TYPE_MASTER)
1369            noctty = 1;
1370    #ifdef TTY_DEBUG_HANGUP
1371        printk("opening %s...", tty_name(tty, buf));
1372    #endif
1373        if (tty->driver.open)
1374            retval = tty->driver.open(tty, filp);
1375        else
1376            retval = -ENODEV;
1377        filp->f_flags = saved_flags;
1378
```

```
1379        if (!retval && test_bit(TTY_EXCLUSIVE, &tty->flags) && !suser( ))
1380            retval = -EBUSY;
1381
1382        if (retval) {
1383    #ifdef TTY_DEBUG_HANGUP
1384            printk("error %d in opening %s...", retval,
1385                   tty_name(tty, buf));
1386    #endif
1387
1388            release_dev(filp);
1389            if (retval != -ERESTARTSYS)
1390                return retval;
1391            if (signal_pending(current))
1392                return retval;
1393            schedule( );
1394            /*
1395             * Need to reset f_op in case a hangup happened.
1396             */
1397            filp->f_op = &tty_fops;
1398            goto retry_open;
1399        }
1400        if (!noctty &&
1401            current->leader &&
1402            !current->tty &&
1403            tty->session == 0) {
1404                task_lock(current);
1405            current->tty = tty;
1406            task_unlock(current);
1407            current->tty_old_pgrp = 0;
1408            tty->session = current->session;
1409            tty->pgrp = current->pgrp;
1410        }
1411        if ((tty->driver.type == TTY_DRIVER_TYPE_SERIAL) &&
1412            (tty->driver.subtype == SERIAL_TYPE_CALLOUT) &&
1413            (tty->count == 1)) {
1414            static int nr_warns;
1415            if (nr_warns < 5) {
1416                printk(KERN_WARNING "tty_io.c: "
1417                    "process %d (%s) used obsolete /dev/%s - "
1418                    "update software to use /dev/ttyS%d\n",
1419                    current->pid, current->comm,
1420                    tty_name(tty, buf), TTY_NUMBER(tty));
1421                nr_warns++;
1422            }
1423        }
1424        return 0;
1425    }
```

每个已打开文件都有个 file 结构作为代表，file 结构中有个 void 指针 private_data，对于常规的文件这个指针很少用到，现在就用来指向目标设备的 tty_struct 数据结构。这样，从当前进程到目标终端设备的连接就建立起来了。同时，tty_struct 结构中有个队列头 tty_files，这里通过 file_move( )将指向该终端设备的 file 结构挂入这个队列，以便在需要时能找到所有与此相连的 file 结构。

前面已经通过相应 tty_ldisc 结构所提供的函数指针调用了与链路规则有关的 open 操作，可是具体的终端类型也可能有需要在打开文件时加以调用的函数(1373 行)。对于用作控制台的虚拟终端，其 tty_driver 数据结构为 console_driver，其 open 函数则为 con_open( )，其代码在 drivers/char/console.c 中：

[tty_open( ) > con_open( )]

```
2304    /*
2305     * Allocate the console screen memory.
2306     */
2307    static int con_open(struct tty_struct *tty, struct file * filp)
2308    {
2309        unsigned int    currcons;
2310        int i;
2311
2312        currcons = MINOR(tty->device) - tty->driver.minor_start;
2313
2314        i = vc_allocate(currcons);
2315        if (i)
2316            return i;
2317
2318        vt_cons[currcons]->vc_num = currcons;
2319        tty->driver_data = vt_cons[currcons];
2320
2321        if (!tty->winsize.ws_row && !tty->winsize.ws_col) {
2322            tty->winsize.ws_row = video_num_lines;
2323            tty->winsize.ws_col = video_num_columns;
2324        }
2325        if (tty->count == 1)
2326            vcs_make_devfs (currcons, 0);
2327        return 0;
2328    }
```

这个函数的作用是为当前的虚拟终端分配缓冲区和有关的数据结构，用来记录该虚拟终端的上下文，有兴趣的读者可自行阅读 vc_allocate( )的代码(drivers/char/console.c)。此外，还要通过 vcs_make_devfs( )在/dev 目录中创建一个 vcs 设备文件(节点)，使应用软件在需要时可以绕过常规的终端设备界面，通过 vcs 设备文件直接访问这个缓冲区。

回到 tty_open( )的代码中，余下的一些代码就留给读者了。

对于运行于“字符模式”的 VGA 卡，应用软件既可以通过常规的终端设备界面访问控制台，也可以通过 vcs 设备文件访问控制台。与此相似，如果 VGA 卡(SVGA 卡)运行于图像模式，则应用软件也可以把控制台作为图像设备打开，绕过常规的终端设备界面而直接访问图像缓冲区，称为“frame

buffer”。这种安排有着特殊的意义，与前述的伪终端设备的使用相结合，就可以让视窗管理进程处理非拼音文字的显示。这样，就有了两种显示汉字的方法，一种是在内核中的虚拟终端(控制台)驱动程序中支持汉字的显示；另一种就是在应用软件中，即在视窗管理进程中支持汉字的显示。

由于篇幅的限制，我们不能在本书中讲到对作为图像设备的控制台，即图像缓冲区的驱动了，有兴趣或需要的读者可以自己阅读有关的代码，这些代码都在 drivers/video 目录下。

8.8 控制台的驱动

在本节中，我们通过 PC 机控制台的读操作来看虚拟终端，包括键盘和 VGA 显示卡的驱动。具体地说，就是从在键盘上输入个别的字符开始，到应用进程从其标准输入通道读取缓冲行为止的过程。虽然这只是个输入过程，但是实际上也包含了对屏幕的操作，因为终端设备的输入操作通常都包括了对输入字符的“回打”(echo)。也许可以说，在字符设备中（除一些网络设备外）控制台的驱动是最复杂的。我们之所以花比较大的篇幅来介绍控制台的驱动，一来正是因为它比较复杂，读者容易遇到问题；二来是因为控制台的驱动，从而对有关代码的理解，对于 Linux 的汉化工作是关键性的。

应用进程通常以其“控制终端”为“标准输入”通道，所以对“标准输入”通道的读操作就转化成对某个虚拟终端的读操作，最后落实成对控制台的读操作。经过读者已经熟知的过程，CPU 通过目标设备的 file_operations 数据结构进入控制台的读操作函数，这就是 tty_read( )，定义于 drivers/char/tty_io.c：

```
645    static ssize_t tty_read(struct file * file, char * buf, size_t count,
646                loff_t *ppos)
647    {
648        int i;
649        struct tty_struct * tty;
650        struct inode *inode;
651
652        /* Can't seek (pread) on ttys.  */
653        if (ppos != &file->f_pos)
654            return -ESPIPE;
655
656        tty = (struct tty_struct *)file->private_data;
657        inode = file->f_dentry->d_inode;
658        if (tty_paranoia_check(tty, inode->i_rdev, "tty_read"))
659            return -EIO;
660        if (!tty || (test_bit(TTY_IO_ERROR, &tty->flags)))
661            return -EIO;
662
663        /* This check not only needs to be done before reading, but also
664           whenever read_chan( ) gets woken up after sleeping, so I've
665           moved it to there.  This should only be done for the N_TTY
666           line discipline, anyway.  Same goes for write_chan( ). -- jlc. */
667    #if 0
       . . . . . .
```

```
678   #endif
679       lock_kernel( );
680       if (tty->ldisc.read)
681           i = (tty->ldisc.read)(tty,file,buf,count);
682       else
683           i = -EIO;
684       unlock_kernel( );
685       if (i > 0)
686           inode->i_atime = CURRENT_TIME;
687       return i;
688   }
```

如前节中所述，在打开(设备)文件的过程中已经设置好了目标设备的 tty_struct 数据结构，并且使 file 结构中的指针 private_data 指向这个数据结构，所以这里 656 行毫不费力就找到了目标设备的 tty_struct 结构。

代表着终端设备的“链路规程”的是 tty_ldisc 数据结构；对于控制台，这就是序号为 N_TTY 的 tty_ldisc 结构 tty_ldisc_N_TTY，这是在初始化时通过 tty_register_ldisc( )向系统登记的。681 行通过它所提供的函数指针 read 进入了驱动程序的下一个子层。这个数据结构定义于 drivers/char/n_tty.c 中：

```
1213   struct tty_ldisc tty_ldisc_N_TTY = {
1214       TTY_LDISC_MAGIC,            /* magic */
1215       "n_tty",                    /* name */
1216       0,                          /* num */
1217       0,                          /* flags */
1218       n_tty_open,                 /* open */
1219       n_tty_close,                /* close */
1220       n_tty_flush_buffer,         /* flush_buffer */
1221       n_tty_chars_in_buffer,      /* chars_in_buffer */
1222       read_chan,                  /* read */
1223       write_chan,                 /* write */
1224       n_tty_ioctl,                /* ioctl */
1225       n_tty_set_termios,          /* set_termios */
1226       normal_poll,                /* poll */
1227       n_tty_receive_buf,          /* receive_buf */
1228       n_tty_receive_room,         /* receive_room */
1229       0                           /* write_wakeup */
1230   };
```

可见，其 read 函数为 read_chan( )，其代码在 drivers/char/n_tty.c 中：

[tty_read( ) > read_chan( )]

```
925   static ssize_t read_chan(struct tty_struct *tty, struct file *file,
926                unsigned char *buf, size_t nr)
927   {
928       unsigned char *b = buf;
```

```
929         DECLARE_WAITQUEUE(wait, current);
930         int c;
931         int minimum, time;
932         ssize_t retval = 0;
933         ssize_t size;
934         long timeout;
935         unsigned long flags;
936 
937     do_it_again:
938 
939         if (!tty->read_buf) {
940             printk("n_tty_read_chan: called with read_buf == NULL?!?\n");
941             return -EIO;
942         }
943 
944         /* Job control check -- must be done at start and after
945            every sleep (POSIX.1 7.1.1.4). */
946         /* NOTE: not yet done after every sleep pending a thorough
947            check of the logic of this change. -- jlc */
948         /* don't stop on /dev/console */
949         if (file->f_dentry->d_inode->i_rdev != CONSOLE_DEV &&
950             file->f_dentry->d_inode->i_rdev != SYSCONS_DEV &&
951             current->tty == tty) {
952             if (tty->pgrp <= 0)
953                 printk("read_chan: tty->pgrp <= 0!\n");
954             else if (current->pgrp != tty->pgrp) {
955                 if (is_ignored(SIGTTIN) ||
956                     is_orphaned_pgrp(current->pgrp))
957                     return -EIO;
958                 kill_pg(current->pgrp, SIGTTIN, 1);
959                 return -ERESTARTSYS;
960             }
961         }
962 
```

读者对上面这一段代码应该不会有困难了，我们把它留给读者结合前几章中的有关内容自己阅读。

[tty_read() > read_chan()]

```
963         minimum = time = 0;
964         timeout = MAX_SCHEDULE_TIMEOUT;
965         if (!tty->icanon) {
966             time = (HZ / 10) * TIME_CHAR(tty);
967             minimum = MIN_CHAR(tty);
968             if (minimum) {
969                 if (time)
970                     tty->minimum_to_wake = 1;
971                 else if (!waitqueue_active(&tty->read_wait) ||
```

```
972                     (tty->minimum_to_wake > minimum))
973                     tty->minimum_to_wake = minimum;
974             } else {
975                 timeout = 0;
976                 if (time) {
977                     timeout = time;
978                     time = 0;
979                 }
980                 tty->minimum_to_wake = minimum = 1;
981             }
982         }
983
984         if (file->f_flags & O_NONBLOCK) {
985             if (down_trylock(&tty->atomic_read))
986                 return -EAGAIN;
987         }
988         else {
989             if (down_interruptible(&tty->atomic_read))
990                 return -ERESTARTSYS;
991         }
992
993         add_wait_queue(&tty->read_wait, &wait);
994         set_bit(TTY_DONT_FLIP, &tty->flags);
```

在“原始”模式和非“规范”模式中，当输入缓冲区中有了最低限度的数据量 minimum_to_wake 时，就要唤醒正在等待着从该设备读出的进程。这里的 965～982 行为其确定一个合适的数值，这个数值一般都是 1。

对从同一终端设备的读出应该是互斥的，所以要放在临界区中。此外，还要在当前进程的系统堆栈中准备下一个 wait_queue_t 数据结构 wait，并把它挂入目标终端的等待队列 read_wait 中，使终端设备的驱动程序在有数据可读时可以唤醒这个进程。当然，也许终端设备的输入缓冲区中现在就有数据，因而根本就不需要进入睡眠，但是那也没有关系，读到了数据之后再把它从队列里摘除就可以了。这里还把 tty->flags 中的一个标志位 TTY_DONT_FLIP 设成 1，读者在后面将会看到这个标志位的意义和作用。

然后就是个 while 循环(drivers/char/n_tty.c)。

[tty_read( ) > read_chan( )]

```
995         while (nr) {
996             /* First test for status change. */
997             if (tty->packet && tty->link->ctrl_status) {
998                 unsigned char cs;
999                 if (b != buf)
1000                    break;
1001                cs = tty->link->ctrl_status;
1002                tty->link->ctrl_status = 0;
```

```
1003                    put_user(cs, b++);
1004                    nr--;
1005                    break;
1006                }
1007                /* This statement must be first before checking for input
1008                   so that any interrupt will set the state back to
1009                   TASK_RUNNING. */
1010                set current_state(TASK_INTERRUPTIBLE);
1011
1012                if (((minimum - (b - buf)) < tty->minimum_to_wake) &&
1013                    ((minimum - (b - buf)) >= 1))
1014                    tty->minimum to_wake = (minimum - (b - buf));
1015
1016                if (!input_available_p(tty, 0)) {
1017                    if (test_bit(TTY OTHER_CLOSED, &tty->flags)) {
1018                        retval = -EIO;
1019                        break;
1020                    }
1021                    if (tty_hung up_p(file))
1022                        break;
1023                    if (!timeout)
1024                        break;
1025                    if (file->f_flags & O_NONBLOCK) {
1026                        retval = -EAGAIN;
1027                        break;
1028                    }
1029                    if (signal_pending(current)) {
1030                        retval = -ERESTARTSYS;
1031                        break;
1032                    }
1033                    clear_bit(TTY_DONT FLIP, &tty->flags);
1034                    timeout = schedule_timeout(timeout);
1035                    set_bit(TTY_DONT_FLIP, &tty->flags);
1036                    continue;
1037                }
```

对于伪终端设备，可以通过系统调用 ioctl( )将主/从两端的通信方式设置成“packet”(信包)模式，因为这两端往往在通过网络连接的不同计算机中。这这种情况下 tty->packet 为 1，而若 tty->link->ctrl_status 非 0 就表示有反映通信链路状态变化的控制信息需要递交，所以要优先读出这些控制信息(一些标志位)。不过这与我们这个情景无关。

指针 b 开始时(928 行)指向用户空间的缓冲区 buf，随着字符的读出而向前推进，所以(b-buf)就是已经读出的字符数。如果 tty->minimum_to_wake 开始时不是 1，那么在读出的过程中会向 1 逼近(1012~1014 行)。

接着通过 input_available_p( )检查输入缓冲区中有否数据(字符)可供读出。在“规范模式”下，检查的是经过加工以后的数量，而在原始模式下则是检查原始字符的数量。这个函数的代码在

drivers/char/n_tty.c 中：

[tty_read( ) > read_chan( ) > input_available_p( )]

```
879    static inline int input_available_p(struct tty_struct *tty, int amt)
880    {
881        if (tty->icanon) {
882            if (tty->canon_data)
883                return 1;
884        } else if (tty->read_cnt >= (amt ? amt : 1))
885            return 1;
886
887        return 0;
888    }
```

如果缓冲区中还没有字符可供读出，则当前进程一般要睡眠等待。到缓冲区中有了可供读出的字符时才会被唤醒。在我们这个情景中，假定此时缓冲区中没有数据，所以当前进程进入了睡眠。

为了更好地把注意力集中在发生键盘中断以后的过程，我们暂且假定中断已经发生而缓冲区中已经有了数据，先来看当前进程被唤醒并被调度运行以后的操作(drivers/char/n_tty.c)。这样，等一下我们就可以集中看从键盘中断开始到唤醒睡眠中的进程为止的过程了。相比之下，那个过程更为重要，也更为复杂。

[tty_read( ) > read_chan( )]

```
1038            current->state = TASK_RUNNING;
1039
1040            /* Deal with packet mode. */
1041            if (tty->packet && b == buf) {
1042                put_user(TIOCPKT_DATA, b++);
1043                nr--;
1044            }
1045
1046            if (tty->icanon) {
1047                /* N.B. avoid overrun if nr == 0 */
1048                while (nr && tty->read_cnt) {
1049                    int eol;
1050
1051                    eol = test_and_clear_bit(tty->read_tail,
1052                            &tty->read_flags);
1053                    c = tty->read_buf[tty->read_tail];
1054                    spin_lock_irqsave(&tty->read_lock, flags);
1055                    tty->read_tail = ((tty->read_tail+1) &
1056                              (N_TTY_BUF_SIZE-1));
1057                    tty->read_cnt--;
1058                    spin_unlock_irqrestore(&tty->read_lock, flags);
1059
```

```
1060                if (!eol || (c != __DISABLED_CHAR)) {
1061                    put_user(c, b++);
1062                    nr--;
1063                }
1064                if (eol) {
1065                    /* this test should be redundant:
1066                     * we shouldn't be reading data if
1067                     * canon_data is 0
1068                     */
1069                    if (--tty->canon_data < 0)
1070                        tty->canon_data = 0;
1071                    break;
1072                }
1073            }
1074        } else {
1075            int uncopied;
1076            uncopied = copy_from_read_buf(tty, &b, &nr);
1077            uncopied += copy_from_read_buf(tty, &b, &nr);
1078            if (uncopied) {
1079                retval = -EFAULT;
1080                break;
1081            }
1082        }
1083
1084        /* If there is enough space in the read buffer now, let the
1085         * low-level driver know. We use n_tty_chars_in_buffer( ) to
1086         * check the buffer, as it now knows about canonical mode.
1087         * Otherwise, if the driver is throttled and the line is
1088         * longer than TTY_THRESHOLD_UNTHROTTLE in canonical mode,
1089         * we won't get any more characters.
1090         */
1091        if (n_tty_chars_in_buffer(tty) <= TTY_THRESHOLD_UNTHROTTLE)
1092            check_unthrottle(tty);
1093
1094        if (b - buf >= minimum)
1095            break;
1096        if (time)
1097            timeout = time;
1098    }
```

当前进程被唤醒时，一般在缓冲区中已经有了数据。我们在这里不关心 packet 模式的操作，所以跳过 1041～1044 行。先看规范模式(1046～1074 行)。在规范模式下，缓冲区中的字符是经过了加工的，要到累积起一个“缓冲行”，即碰到“\n”字符时才会唤醒等待读出的进程，此时 tty->read_cnt 表示缓冲行中的字符个数。另一方面，缓冲区 tty->read_buf[ ]是按环形缓冲区使用的(见 1055 行)，tty->read_tail 指向当前可供读出的第一个字符。前一节中曾经讲到，tty_struct 结构中的 read_flags 是个位图，如果这个位图中对应于 tty->read_tail 这一位为 1，则表示这个位置上已经是缓冲行的终点，以后的数据已经属

于另一缓冲行。所以，只要还没有到达终点，就逐个字符地调用 put_user( )将其复制到用户空间去。这个过程要循环到已经满足了要求或者缓冲区中已经没有字符可读时才结束。这里的__DISABLED_CHAR 就是“\0”，缓冲行中的最后一个字符如果是__DISABLED_CHAR 就不复制到用户空间。这个字符定义于 include/linux/tty.h：

```
125    /*
126     * This character is the same as  POSIX VDISABLE: it cannot be used as
127     * a c cc[ ] character, but indicates that a particular special character
128     * isn't in use (eg VINTR has no character etc)
129     */
130    #define   DISABLED CHAR '\0'
```

再看非规范模式(1074～1082 行)。缓冲区同样也是 tty->read_buf[ ]，也按环形缓冲区使用，这些都一样，但是缓冲区中的字符是未经加工的，也无所谓“缓冲行”。另一方面，对于原始模式并没有不让把“\0”复制到用户空间的规定，所以这里通过 copy_from_read_buf( )进行成片的复制，以加快速度，这个函数的代码在 drivers/char/n_tty.c 中：

[tty_read( ) > read_chan( ) > copy_from_read_buf( )]

```
890    /*
891     * Helper function to speed up read chan.  It is only called when
892     * ICANON is off; it copies characters straight from the tty queue to
893     * user space directly.  It can be profitably called twice; once to
894     * drain the space from the tail pointer to the (physical) end of the
895     * buffer, and once to drain the space from the (physical) beginning of
896     * the buffer to head pointer.
897     */
898    static inline int copy_from_read_buf(struct tty_struct *tty,
899                          unsigned char **b,
900                          size_t *nr)
901
902    {
903        int retval;
904        ssize t n;
905        unsigned long flags;
906
907        retval = 0;
908        spin_lock_irqsave(&tty->read_lock, flags);
909        n = MIN(*nr, MIN(tty->read_cnt, N_TTY_BUF_SIZE - tty->read_tail));
910        spin_unlock_irqrestore(&tty->read_lock, flags);
911        if (n) {
912            mb( );
913            retval = copy_to_user(*b, &tty->read_buf[tty->read_tail], n);
914            n -= retval;
915            spin_lock_irqsave(&tty->read_lock, flags);
916            tty->read_tail = (tty->read_tail + n) & (N_TTY_BUF_SIZE-1);
```

```
917             tty->read_cnt -= n;
918             spin_unlock_irqrestore(&tty->read_lock, flags);
919             *b += n;
920             *nr -= n;
921         }
922         return retval;
923     }
```

由于缓冲区是环形的，缓冲着的字符有可能分成两段，所以要调用 copy_from_read_buf( )两次。

缓冲区的大小总是有限的，如果从键盘打入字符的速度很快，而应用进程又来不及从缓冲区读出，则底层的驱动程序(主要是中断服务程序)可能已经因为缓冲区已满而暂时把“阀门”关闭了。现在，如果缓冲区中剩余的字符数量降到了“低水位”TTY_THRESHOLD_UNTHROTTLE 以下，则要通过 check_unthrottle( )再打开阀门，这个函数的代码也在 drivers/char/n_tty.c 中：

[tty_read( ) > read_chan( ) > check_unthrottle( )]

```
105     /*
106      * Check whether to call the driver.unthrottle function.
107      * We test the TTY_THROTTLED bit first so that it always
108      * indicates the current state.
109      */
110     static void check_unthrottle(struct tty_struct * tty)
111     {
112         if (tty->count &&
113             test_and_clear_bit(TTY_THROTTLED, &tty->flags) &&
114             tty->driver.unthrottle)
115             tty->driver.unthrottle(tty);
116     }
```

除了清除代表着阀门的标志位 TTY_THROTTLED 以外，还可能要调用一个函数，具体取决于终端的 tty_driver 数据结构。对于控制台终端的 tty_driver 数据结构 console_driver，这个函数是 con_unthrottle( )，其代码在 drivers/char/console.c 中：

[tty_read( ) > read_chan( ) > check_unthrottle( ) > con_unthrottle( )]

```
2256    static void con_unthrottle(struct tty_struct *tty)
2257    {
2258        struct vt_struct *vt = (struct vt_struct *) tty->driver_data;
2259
2260        wake_up_interruptible(&vt->paste_wait);
2261    }
```

可见，目的只是唤醒可能在等待着要把数据写入缓冲区的进程。

回到 read_chan( )的代码中继续往下看(drivers/char/n_tty.c)：

[tty_read( ) > read_chan( )]

```
1099        clear_bit(TTY_DONT_FLIP, &tty->flags);
1100        up(&tty->atomic_read);
1101        remove_wait_queue(&tty->read_wait, &wait);
1102
1103        if (!waitqueue_active(&tty->read_wait))
1104            tty->minimum_to_wake = minimum;
1105
1106        current->state = TASK_RUNNING;
1107        size = b - buf;
1108        if (size) {
1109            retval = size;
1110            if (nr)
1111                    clear_bit(TTY_PUSH, &tty->flags);
1112        } else if (test_and_clear_bit(TTY_PUSH, &tty->flags))
1113             goto do_it_again;
1114
1115        return retval;
1116    }
```

当前进程既已读到了所要求的输入，需要放在临界区中进行的操作就完成了。这里的指针 buf 指向用户空间的缓冲区， 而 b 指出向该缓冲区中的下一个空闲位置，所以 b-buf 就是已经读入该缓冲区中的字符数量。参数 nr 只是表明用户空间缓冲区的大小，即读出字符数量的上限，在规范模式下实际读出的字符数取决于具体的缓冲行。

标志位 TTY_PUSH 是由低层驱动程序在读到一个 EOF 字符并将其放入缓冲区时设置成 1 的，表示要让用户尽快把缓冲区的内容读走。如果此后从缓冲区读出了缓冲区中的所有字符，就把这个标志位清 0 并结束整个读出操作；否则，就说明还要继续从缓冲区读。所以如果本次读操作实际并未读出，则不让它结束，通过 1113 行的 goto 语句转到前面(937 行)的 do_it_again 处。如果本次读操作多少已经读出了若干字符则容许下次再读。

一般而言，一个典型的读终端过程可以分成下列三部分，或者这三部分的循环：

(1) 当前进程企图从终端的缓冲区读出，但是因为缓冲区中尚无足够字符供读出而受阻，进入睡眠；

(2) 然后，当使用者在键盘上输入字符，低层驱动程序将足够的字符写入缓冲区以后，就把睡眠中的进程唤醒；

(3) 睡眠中的进程被唤醒以后，继续完成读出。

我们在上面阅读的是这个过程中的第一、三两部分的代码，阅读时只是假定第二部分已经发生。实际上，第二部分才是设备驱动程序的主体，对于终端设备更是如此，下面我们就来看这一部分的代码。

使用者在键盘上按一个键，就产生了一个中断请求，CPU 在响应中断时便进入键盘的中断服务程序 keyboard_interrupt( )。人们往往以为键盘是很简单的设备，但是实际上 PC 键盘的结构和操作都不简单。我们以字符“A”为例来说明键盘的操作。当在键盘上按下一个键时，键盘立即就向母板发出一个字节的代码，称为“键盘扫描码”。具体的值取决于键的位置，“A”键的键盘扫描码为 0x1c。母板

上的键盘接口在接收到这个字节以后就把它转换成一种“系统扫描码”，并将其存入控制器的内部缓冲区，然后便向 CPU 发出一个中断请求。对于“A”键，其系统扫描码为 0x1e。当 CPU 从键盘接口的数据寄存器读时，读出的就是系统扫描码。然后，当放开“A”键时，键盘又要向母板发出键盘扫描码，而这一次的键盘扫描码是两个字节，第一个总是 0xf0，表示是放开一个键；然后是 0x1c，表示是哪一个键。同样，母板上的键盘接口也要把它转换成系统扫描码、也要向 CPU 发出中断请求，但是系统扫描码仍是一个字节，只是在 0x1e 的基础上把最高位设成 1，变成了 0x9e。这样，在 CPU 从键盘接口读出的单字节系统扫描码中，其最高位表示按下或放开，而低 7 位则与具体的键相对应。不光对普通的字符键是如此，对功能键和控制键也是一样。例如，左右两个 Shift 键的系统扫描码分别为 0x2a 和 0x36。这样，以输入一个小写的“a”为例，CPU 实际上可能要发生 4 次中断、要依次从键盘接口读出 4 个字节的系统扫描码，例如：0x36，0x1e，0x9e，0xb6。至于把这个序列解释成什么，那就是软件的事了；如果解释成 ASCII 码，而键盘又没有锁定在大写状态，那就是“a”。由此可见，键盘接口向 CPU 发出的中断请求并不意味着从键盘接收到了一个字符，而只是意味着键盘上发生了某种事件。此外，上面只是一般而言，实际上还有不少例外。其中最重要的是所谓“扩充键”。早期的 PC 键盘上只有 83 个键，后来扩充到了 101 键或 104 键，例如右边的 Ctrl 键就是一个扩充键。当按下或放开扩充键时，键盘扫描码和系统扫描码都以一个 0xe0 字节开头，所以按下右边 Ctrl 键时的键盘扫描码是[0xe0，0x14]；放开时为[0xe0，0xf0，0x14]；相应的系统扫描码则为[0xe0，0x1d]以及[0xe0，0x9d]。

把这些事件和产生的代码列成表可以看得更清楚一些：

事件	键盘扫描码	系统扫描码
按下“A”键	0x1c	0x1e
放开“A”键	0xf0，0x1c	0x9e
按下右 Shift 键	0x59	0x36
放开右 Shift 键	0xf0，0x59	0xb6
按下右 Ctrl 键	0xe0，0x14	0xe0，0x1d
放开右 Ctrl 键	0xe0，0xf0，0x14	0xe0，0x9d

其他还有一些例外，主要与控制键和锁定键有关，我们在这里就不详细介绍了。

之所以把键盘扫描码转换成系统扫描码，是为了建立起一个统一的扫描码界面。这种转换是由键盘接口完成的，所以不占用 CPU 的时间。不过，有需要时也可以通过键盘控制器关闭这种转换。显然，对于软件而言键盘扫描码是不可见的，所以我们在下面讲到“扫描码”时都是指系统扫描码。此外，按下一个键时产生的扫描码常常称为“make code”，而放开一个键时产生的扫描码则常常称为“break code”。

函数 keyboard_interrupt( )的代码在 drivers/char/pc_keyb.c 中：

```
479	static void keyboard_interrupt(int irq, void *dev_id, struct pt_regs *regs)
480	{
481	#ifdef CONFIG_VT
482		kbd_pt_regs = regs;
483	#endif
```

```
484
485         spin_lock_irq(&kbd_controller_lock);
486         handle_kbd_event( );
487         spin_unlock_irq(&kbd_controller_lock);
488     }
```

为防止不同的 CPU 同时对键盘操作(多处理器系统中)，内核中为键盘操作设立了一把锁，即 kbd_controller_lock，以保证键盘操作的互斥性。函数 handle_kbd_event( )的代码在同一文件中：

[keyboard_interrupt( ) > handle_kbd_event( )]

```
446     static unsigned char handle_kbd_event(void)
447     {
448         unsigned char status = kbd_read_status( );
449         unsigned int work = 10000;
450
451         while ((--work > 0) && (status & KBD_STAT_OBF)) {
452             unsigned char scancode;
453
454             scancode = kbd_read_input( );
455
456             /* Error bytes must be ignored to make the
457                Synaptics touchpads compaq use work */
458     #if 1
459             /* Ignore error bytes */
460             if (!(status & (KBD_STAT_GTO | KBD_STAT_PERR)))
461     #endif
462             {
463                 if (status & KBD_STAT_MOUSE_OBF)
464                     handle_mouse_event(scancode);
465                 else
466                     handle_keyboard_event(scancode);
467             }
468
469             status = kbd_read_status( );
470         }
471
472         if (!work)
473             printk(KERN_ERR "pc_keyb: controller jammed (0x%02X).\n", status);
474
475         return status;
476     }
```

就像其他设备一样，键盘接口上也有“控制/状态寄存器”和“数据寄存器”。首先要读出状态寄存器，看看发生了什么事。状态寄存器中有个标志位 KBD_STAT_OBF(OBF 表示“Output Buffer Full”)，当这个标志位为 1 时就表示键盘的内部缓冲区中有数据，可以通过数据寄存器读出。这里的 kbd_read_status( )和 kbd_read_input( )都是定义于 include/asm-i386/keyboard.h 中的宏操作：

```
48      #define kbd_read_input( ) inb(KBD_DATA_REG)
49      #define kbd_read_status( ) inb(KBD_STATUS_REG)
```

前面讲过，从 PC 键盘读入的是扫描码，代表着相应的键在键盘上的位置以及键的状态，而与具体文字的编码无关，因此要根据一定的规则将扫描码转换成相应的代码。同时，缓冲区中可能有不止一个的字节，因而要通过一个 while 循环从缓冲区逐个字节地读出。鼠标器通常与键盘共用同一个接口，所以还要根据状态寄存器的内容确定数据的来源，我们在这里假定确实来自键盘。从键盘的数据寄存器读出了全部字符以后，状态寄存器中的标志位 KBD_STAT_OBF 就变成 0。如果读了 10000 次以后这个标志位还是 1，那当然是有问题了。对从键盘读入的每个字符都通过 handle_keyboard_event( )进一步处理，其代码也在 drivers/char/pc_keyb.c 中：

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_keyboard_event( )]

```
429     static inline void handle_keyboard_event(unsigned char scancode)
430     {
431     #ifdef CONFIG_VT
432         kbd_exists = 1;
433         if (do_acknowledge(scancode))
434             handle_scancode(scancode, !(scancode & 0x80));
435     #endif
436         tasklet_schedule(&keyboard_tasklet);
437     }
438
```

键盘并不是一种“只读”的设备，对键盘也有输出操作。键盘在收到数据后都要送回一个 KBD_REPLY_ACK（定义为 0xfa）予以确认，或送回一个 KBD_REPLY_RESEND（定义为 0xfe）要求重发，而 CPU 必须将其与正常的输入区分开来。为了这个目的，内核中设了一个全局量 reply_expected，每当发送一个字节给键盘时就将 reply_expected 设成 1，而在 handle_keyboard_event( )中若发现 reply_expected 为 1 就要把输入丢弃。这就是这里调用 do_acknowledge( )的目的，其代码在 drivers/char/pc_keyb.c 中：

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_keyboard_event( ) > do_acknowledge( )]

```
265     static int do_acknowledge(unsigned char scancode)
266     {
267         if (reply_expected) {
268           /* Unfortunately, we must recognise these codes only if we know they
269            * are known to be valid (i.e., after sending a command), because there
270            * are some brain-damaged keyboards (yes, FOCUS 9000 again) which have
271            * keys with such codes :(
272            */
273             if (scancode == KBD_REPLY_ACK) {
274                 acknowledge = 1;
275                 reply_expected = 0;
276                 return 0;
```

```
277             } else if (scancode == KBD_REPLY_RESEND) {
278                 resend = 1;
279                 reply_expected = 0;
280                 return 0;
281             }
282             /* Should not happen... */
283     #if 0
284             printk(KERN_DEBUG "keyboard reply expected - got %02x\n",
285                    scancode);
286     #endif
287         }
288         return 1;
289     }
```

对扫描码的实际处理是由 handle_scancode( )完成的，其代码在 drivers/char/keyboard.c 中。这个函数比较长，我们分段阅读：

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_keyboard_event( ) > handle_scancode( )]

```
201     void handle_scancode(unsigned char scancode, int down)
202     {
203         unsigned char keycode;
204         char up_flag = down ? 0 : 0200;
205         char raw_mode;
206 
207         pm_access(pm_kbd);
208 
209         do_poke_blanked_console = 1;
210         tasklet_schedule(&console_tasklet);
211         add_keyboard_randomness(scancode | up_flag);
212 
213         tty = ttytab? ttytab[fg_console]: NULL;
214         if (tty && (!tty->driver_data)) {
215             /*
216              * We touch the tty structure via the the ttytab array
217              * without knowing whether or not tty is open, which
218              * is inherently dangerous.  We currently rely on that
219              * fact that console_open sets tty->driver_data when
220              * it opens it, and clears it when it closes it.
221              */
222             tty = NULL;
223         }
224         kbd = kbd_table + fg_console;
225         if ((raw_mode = (kbd->kbdmode == VC_RAW))) {
226             put_queue(scancode | up_flag);
227             /* we do not return yet, because we want to maintain
228                the key_down array, so that we have the correct
```

```
229                 values when finishing RAW mode or when changing VT's */
230         }
231
232         /*
233          *  Convert scancode to keycode
234          */
235         if (!kbd_translate(scancode, &keycode, raw_mode))
236             return;
237
```

参数 down 为 1 表示扫描码的最高位为 0，表示键处于按下状态。局部量 up_flag 实际上相当于把扫描码的最高位抽了出来。

这里的 pm_access( )是为电源管理留下的一个口子，其意图是在人机界面上长时间没有活动以后就使显示器进入省电模式，然后一旦有键盘输入时就恢复到正常运行。不过，在目前的代码中这是一个空函数。

我们在第 3 章中讲过，中断服务程序不宜太长，有些可能比较费时的操作应该放在比较宽松的 bh 函数或 tasklet 中完成。对虚拟控制台的切换就是这样一种操作，所以控制台操作有一个 tasklet，那就是 console_tasklet( )。这个 tasklet 是准备在执行完键盘中断服务程序以后，在从中断响应返回之前执行的，这里要通过 tasklet_schedule( )，将其挂入相应的队列中。这里还调用了一个函数 add_keyboard_randomness( )，这是借键盘输入的随机性加大系统中伪随机数的随机性，读者在前几章中也看到过类似的运用。

PC 机的控制台终端由显示器(图形卡)和键盘两部分构成，所以除 tty_struct 数据结构外还有个 kbd_struct 数据结构。同时，物理的显示器和键盘又可用于多个虚拟终端，通过 Alt 键与功能键的组合来切换。显然，每个虚拟终端都应该有自己的 tty_struct 结构和 kbd_struct 结构。为此，内核中设立了 ttytab[ ]和 kbd_table[ ]两个结构数组，而全局量 fg_console 则记录着当前的“前台”虚拟终端号。同时，为便于后面的处理，又设立了 tty 和 kbd 两个全局量，使它们分别指向“前台”虚拟终端的 tty_struct 结构和 kbd_struct 结构。

如果前台终端的键盘工作于“原始”模式 VC_RAW (可以通过系统调用 ioctl( )设置), 那就直接把扫描码放到键盘的接收队列中，否则就要将扫描码转换成“键码”后才放到队列中。所谓“原始”(raw)模式对于不同的层次有不同的意义，键盘的原始模式有两种，其中最原始的就是 VC_RAW，表示直接把扫描码送给应用层。

我们先看转换扫描码的过程 kbd_translate( )，这个函数在 include/asm-i386/keyboard.h 中定义成 pckbd_translate( )：

```
34      #define kbd_translate          pckbd_translate
```

这就是 PC 键盘的键码转换函数，其代码在 drivers/char/pc_keyb.c 中：

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_keyboard_event( ) > handle_scancode( ) > pckbd_translate( )]

```
291     int pckbd_translate(unsigned char scancode, unsigned char *keycode,
```

```
292                 char raw_mode)
293     {
294         static int prev_scancode;
295
296         /* special prefix scancodes.. */
297         if (scancode == 0xe0 || scancode == 0xe1) {
298             prev_scancode = scancode;
299             return 0;
300         }
301
302         /* 0xFF is sent by a few keyboards, ignore it. 0x00 is error */
303         if (scancode == 0x00 || scancode == 0xff) {
304             prev_scancode = 0;
305             return 0;
306         }
307
308         scancode &= 0x7f;
309
310         if (prev_scancode) {
311           /*
312            * usually it will be 0xe0, but a Pause key generates
313            * e1 1d 45 e1 9d c5 when pressed, and nothing when released
314            */
315           if (prev_scancode != 0xe0) {
316               if (prev_scancode == 0xe1 && scancode == 0x1d) {
317               prev_scancode = 0x100;
318               return 0;
319               } else if (prev_scancode == 0x100 && scancode == 0x45) {
320               *keycode = E1_PAUSE;
321               prev_scancode = 0;
322               } else {
323     #ifdef KBD_REPORT_UNKN
324               if (!raw_mode)
325                 printk(KERN_INFO "keyboard: unknown e1 escape sequence\n");
326     #endif
327               prev_scancode = 0;
328               return 0;
329               }
330           } else {
331               prev_scancode = 0;
332               /*
333                *  The keyboard maintains its own internal caps lock and
334                *  num lock statuses. In caps lock mode E0 AA precedes make
335                *  code and E0 2A follows break code. In num lock mode,
336                *  E0 2A precedes make code and E0 AA follows break code.
337                *  We do our own book-keeping, so we will just ignore these.
338                */
339               /*
```

```
340                 * For my keyboard there is no caps lock mode, but there are
341                 * both Shift-L and Shift-R modes. The former mode generates
342                 * E0 2A / E0 AA pairs, the latter E0 B6 / E0 36 pairs.
343                 * So, we should also ignore the latter. - aeb@cwi.nl
344                 */
345                if (scancode == 0x2a || scancode == 0x36)
346              return 0;
347
348                if (e0_keys[scancode])
349              *keycode = e0_keys[scancode];
350                else {
351    #ifdef KBD_REPORT_UNKN
352                if (!raw_mode)
353                  printk(KERN_INFO "keyboard: unknown scancode e0 %02x\n",
354                      scancode);
355    #endif
356                return 0;
357                }
358            }
359        } else if (scancode >= SC_LIM) {
```

首先，如前所述，如果读入的扫描码是 0xe0（或 0xe1），那就是扩充键的前缀码。此时的扫描码是个序列，所以需要为之实现一种“有限状态机”，全局量 prev_scancode 就是用于这个目的。这里先把前缀码作为一种状态保存在 prev_scancode 中，并返回 0，表示这个字节应予丢弃，而 handle_scancode( ) 也就随之返回(236 行)。此外，0x00 和 0xff 不是有效的扫描码，也要丢弃。

对于扫描码本身的处理因是否扩充键而异。如果 prev_scancode 非 0，那就说明在此之前的字节是个前缀码，此时又要看前缀码是否为 0xe0。

前缀码 0xe1 是个特例，在按下或放开 Pause 键的时候，键盘向主机发出一个三字节序列[0xe1，0x1d，0x45]或[0xe1，0x1d，0xc5]。所以，代码中为这个序列设置了个中间状态 0x100，316～329 行就是对这个特殊序列的检验。如果三个字节都对，那就是 E1_PAUSE，否则就予以丢弃。除 Pause 键以外，其他扩充键的前缀码都是 0xe0。前面讲到，左右两个 Shift 键并非扩充键，但是有些键盘在 NumLock 或 CapsLock 状态下操作左右 Shift 键时会把它们当成扩充键。由于 Linux 内核自己维持各种锁定状态，所以丢弃作为扩充键的左右 Shift 键扫描码(345～346 行)。

对于扩充键，从扫描码到键码的转换由定义于 drivers/char/pc_keyb.c 的数组 e0_keys[ ]提供：

```
227    static unsigned char e0_keys[128] = {
228      0, 0, 0, 0, 0, 0, 0, 0,                              /* 0x00-0x07 */
229      0, 0, 0, 0, 0, 0, 0, 0,                              /* 0x08-0x0f */
230      0, 0, 0, 0, 0, 0, 0, 0,                              /* 0x10-0x17 */
231      0, 0, 0, 0, E0_KPENTER, E0_RCTRL, 0, 0,              /* 0x18-0x1f */
232      0, 0, 0, 0, 0, 0, 0, 0,                              /* 0x20-0x27 */
233      0, 0, 0, 0, 0, 0, 0, 0,                              /* 0x28-0x2f */
234      0, 0, 0, 0, 0, E0_KPSLASH, 0, E0_PRSCR,              /* 0x30-0x37 */
235      E0_RALT, 0, 0, 0, 0, E0_F13, E0_F14, E0_HELP,        /* 0x38-0x3f */
```

```
236        E0_DO, E0_F17, 0, 0, 0, 0, E0_BREAK, E0_HOME,       /* 0x40-0x47 */
237        E0_UP, E0_PGUP, 0, E0_LEFT, E0_OK, E0_RIGHT, E0_KPMINPLUS, E0_END, /* 0x48-0x4f */
238        E0_DOWN, E0_PGDN, E0_INS, E0_DEL, 0, 0, 0, 0,       /* 0x50-0x57 */
239        0, 0, 0, E0_MSLW, E0_MSRW, E0_MSTM, 0, 0,           /* 0x58-0x5f */
240        0, 0, 0, 0, 0, 0, 0, 0,                             /* 0x60-0x67 */
241        0, 0, 0, 0, 0, 0, 0, E0_MACRO,                      /* 0x68-0x6f */
242        0, 0, 0, 0, 0, 0, 0, 0,                             /* 0x70-0x77 */
243        0, 0, 0, 0, 0, 0, 0, 0                              /* 0x78-0x7f */
244     };
```

例如，右 Ctrl 键的扫描码为[0xe0, 0x1d]，所以用 0x1d 为下标从表中查得其键码 E0_RCTRL。又如 PageUp 键的扫描码为[0xe0, 0x49]，所以用 0x49 为下标从表中查得其键码 E0_PGUP。这些键码也都定义于同一文件(drivers/char/pc_keyb.c)中：

```
131     /*
132      * Translation of escaped scancodes to keycodes.
133      * This is now user-settable.
134      * The keycodes 1-88,96-111,119 are fairly standard, and
135      * should probably not be changed - changing might confuse X.
136      * X also interprets scancode 0x5d (KEY_Begin).
137      *
138      * For 1-88 keycode equals scancode.
139      */
140
141     #define E0_KPENTER 96
142     #define E0_RCTRL    97
143     #define E0_KPSLASH 98
144     #define E0_PRSCR    99
145     #define E0_RALT     100
146     #define E0_BREAK    101  /* (control-pause) */
147     #define E0_HOME     102
148     #define E0_UP       103
149     #define E0_PGUP     104
150     #define E0_LEFT     105
151     #define E0_RIGHT    106
152     #define E0_END      107
153     #define E0_DOWN     108
154     #define E0_PGDN     109
155     #define E0_INS      110
156     #define E0_DEL      111
157
158     #define E1_PAUSE    119
```

数组 e0_keys[ ]的内容是可以通过系统调用 ioctl( )设置的，但是这里的注释说改了以后可能会使 X window 软件弄糊涂。这也部分地回答了读者可能会有的疑问，就是为什么需要“键码”。

再看不带前缀的扫描码，即非扩充键的扫描码。

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_keyboard_event( ) > handle_scancode( ) > pckbd_translate( )]

```
359        } else if (scancode >= SC_LIM) {
360            /* This happens with the FOCUS 9000 keyboard
361               Its keys PF1..PF12 are reported to generate
362               55 73 77 78 79 7a 7b 7c 74 7e 6d 6f
363               Moreover, unless repeated, they do not generate
364               key-down events, so we have to zero up_flag below */
365            /* Also, Japanese 86/106 keyboards are reported to
366               generate 0x73 and 0x7d for \ - and \ | respectively. */
367            /* Also, some Brazilian keyboard is reported to produce
368               0x73 and 0x7e for \ ? and KP-dot, respectively. */
369
370          *keycode = high_keys[scancode - SC_LIM];
371
372          if (!*keycode) {
373              if (!raw_mode) {
374    #ifdef KBD_REPORT_UNKN
375              printk(KERN_INFO "keyboard: unrecognized scancode (%02x)"
376                 " - ignored\n", scancode);
377    #endif
378              }
379              return 0;
380          }
381        } else
382          *keycode = scancode;
383        return 1;
384    }
```

常数 SC_LIM 的值为 89，即 0x59。数值小于 SC_LIM 的扫描码与键码相同而无需转换(382 行)；否则便由数组 high_keys[ ]提供转换，转换时以(scancode - SC_LIM)为下标。这个数组的定义也在 drivers/char/pc_keyb.c 中：

```
194    static unsigned char high_keys[128 - SC_LIM] = {
195      RGN1, RGN2, RGN3, RGN4, 0, 0, 0,                       /* 0x59-0x5f */
196      0, 0, 0, 0, 0, 0, 0, 0,                                /* 0x60 0x67 */
197      0, 0, 0, 0, 0, FOCUS PF11, 0, FOCUS_PF12,              /* 0x68-0x6f */
198      0, 0, 0, FOCUS_PF2, FOCUS_PF9, 0, 0, FOCUS_PF3,        /* 0x70 0x77 */
199      FOCUS_PF4, FOCUS_PF5, FOCUS_PF6, FOCUS_PF7,            /* 0x78-0x7b */
200      FOCUS_PF8, JAP_86, FOCUS_PF10, 0                       /* 0x7c-0x7f */
201    };
```

这个范围中的大多是功能键，其键码的数值在 89～127 范围中，如 FOCUS_PF2 就是 PF2 键的键码（FOCUS_PF1 的数值为 85，与扫描码相同，所以不在这个数组中）。现在可以比较完整地回答为什么需要将扫描码转换成键码的问题了。带前缀 0xe0 和不带前缀的扫描码占据着两块 128 字节(实际上

只有 126 字节)的区间，但是实际上两个区间都是很“稀疏”的。就键盘的大小来说，现在还在 120 键以下，如果一键一码则只需要 120 字节以下，完全可以合并到同一个 126 字节的区间中去。可是，又不宜简单地把前缀丢弃了事，因为那样至少会造成功能键与普通字符键穿插在一起而带来不便。比较好的办法当然是统一编排一下、定义出一个标准的键码，使功能键的代码都集中在一起，例如上述的 96～111。这样，不管是 PC 键盘也好、Macintoshi 键盘也好，一旦转换到键码就都一样，与具体的键盘无关了。这里还要指出，键码仍然只是一种中性的代码，而与具体的语言文字无关。

回到 handle_scancode( )的代码中继续往下看(drivers/char/keyboard.c)：

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_keyboard_event( ) > handle_scancode( )]

```
238 	/*
239 	 * At this point the variable `keycode' contains the keycode.
240 	 * Note: the keycode must not be 0 (++Geert: on m68k 0 is valid).
241 	 * We keep track of the up/down status of the key, and
242 	 * return the keycode if in MEDIUMRAW mode.
243 	 */
244 
245 	if (up_flag) {
246 		rep = 0;
247 		if(!test_and_clear_bit(keycode, key_down))
248 		    up_flag = kbd_unexpected_up(keycode);
249 	} else
250 		rep = test_and_set_bit(keycode, key_down);
251 
252 #ifdef CONFIG_MAGIC_SYSRQ		/* Handle the SysRq Hack */
253 	if (keycode == SYSRQ_KEY) {
254 		sysrq_pressed = !up_flag;
255 		return;
256 	} else if (sysrq_pressed) {
257 		if (!up_flag) {
258 			handle_sysrq(kbd_sysrq_xlate[keycode], kbd_pt_regs, kbd, tty);
259 			return;
260 		}
261 	}
262 #endif
263 
264 	if (kbd->kbdmode == VC_MEDIUMRAW) {
265 		/* soon keycodes will require more than one byte */
266 		put_queue(keycode + up_flag);
267 		raw_mode = 1;	/* Most key classes will be ignored */
268 	}
269 
```

前面讲过，局部量 up_flag 的值表明相应的键处于按下(down)还是放开(up)状态。当然，键盘上绝大部分的键都处于放开状态，但是也可能同时有几个键处于按下状态，所以代码中使用一个全局的位图 key_down 来记录各个键的状态。如果读入的扫描码表明相应的键处于按下状态，那就把位图

key_down 中的对应位设成 1；进一步，如果这一位原来就已经是 1，那就说明使用者按下这个键不放，所以键盘开始了自动重复功能(250 行)。反之，如果读入的扫描码表明相应的键处于放开状态，那就把位图中的对应位设成 0；如果原来就是 0，就说明至少是漏了一个扫描码，所以说是 kbd_unexpected_up( )。

作为编译选择项，对 SysRq 键可以定义一些特殊的操作，不过我们在这里不感兴趣。

如果键盘的运行模式是 VC_MEDIUMRAW，或者说是“半生半熟”、“半原始”，那么至此已可以把转换后的键码放入接收队列了(266 行)。这样，如果键盘运行于两种原始模式之一，应用进程从键盘(控制台)读到的就是扫描码或键码。在前一节中讲到的伪终端加窗口管理进程的系统结构中，只有窗口管理进程才会直接从键盘读，所以可以在这个进程的软件中处理从扫描码或键码向具体语言文字(如汉字)编码的转换。当然，也可以不让键盘运行于原始模式，而在内核中完成这最后一步转换。读者还应注意，这里所说的原始模式是对最底层的键盘驱动而言，区别在于是否把扫描码转换成目标文字的代码(如 ASCII)，这与前面在“终端”一层上的非规范模式不同，那种“原始”模式后面我们还要讲到。

我们再往下看对键盘输入的最后一步转换，这一次是要转换成 ASCII 码(或其他拼音文字的代码，下同)。在正常模式下运行的键盘要到把输入转换成 ASCII 码以后才放入接收队列，有的则根本不放入接收队列。

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_keyboard_event( ) > handle_scancode( )]

```
270         /*
271          * Small change in philosophy: earlier we defined repetition by
272          *    rep = keycode == prev_keycode;
273          *    prev_keycode = keycode;
274          * but now by the fact that the depressed key was down already.
275          * Does this ever make a difference? Yes.
276          */
277 
278         /*
279          *  Repeat a key only if the input buffers are empty or the
280          *  characters get echoed locally. This makes key repeat usable
281          *  with slow applications and under heavy loads.
282          */
283         if (!rep ||
284             (vc_kbd_mode(kbd,VC_REPEAT) && tty &&
285              (L_ECHO(tty) || (tty->driver.chars_in_buffer(tty) == 0)))) {
286             u_short keysym;
287             u_char type;
288 
289             /* the XOR below used to be an OR */
290             int shift_final = (shift_state | kbd->slockstate) ^
291                 kbd->lockstate;
292             ushort *key_map = key_maps[shift_final];
293 
294             if (key_map != NULL) {
295                 keysym = key_map[keycode];
```

```
296                 type = KTYP(keysym);
297
298                 if (type >= 0xf0) {
299                     type -= 0xf0;
300                     if (raw_mode && ! (TYPES_ALLOWED_IN_RAW_MODE & (1 << type)))
301                     return;
302                     if (type == KT_LETTER) {
303                     type = KT_LATIN;
304                     if (vc_kbd_led(kbd, VC_CAPSLOCK)) {
305                         key_map = key_maps[shift_final ^ (1<<KG_SHIFT)];
306                         if (key_map)
307                           keysym = key_map[keycode];
308                     }
309                     }
310                     (*key_handler[type])(keysym & 0xff, up_flag);
311                     if (type != KT_SLOCK)
312                       kbd->slockstate = 0;
313                 } else {
314                     /* maybe only if (kbd->kbdmode == VC_UNICODE) ? */
315                     if (!up_flag && !raw_mode)
316                       to_utf8(keysym);
317                 }
318             } else {
319                 /* maybe beep? */
320                 /* we have at least to update shift_state */
321     #if 1           /* how? two almost equivalent choices follow */
322                 compute_shiftstate( );
323                 kbd->slockstate = 0; /* play it safe */
324     #else
    . . . . . . .
329     #endif
330             }
331         }
332     }
```

对于正常状态下产生的键盘输入，即 rep 为 0 时，这一段代码的执行是无条件的。而对于因按下键不放而自动重复产生的键盘输入，则是有条件的，条件是键盘运行于 VC_REPEAT 模式，即允许自动重复，并且终端运行于“echo”模式，或者输入缓冲区已经空了(以前的输入都已被读走)。如果不满足这些条件，键盘输入就被丢弃了。

从键码向目标码 keysym 的转换也是通过数组实现的，以键码的数值为下标，相应的元素就给出目标码。由于键码的数值不超过 127，所以数组的大小为 128。数组的类型为 16 位无符号短整数，其高 8 位表示键的类型，低 8 位则为具体的代码。Linux 内核的“母语”是英语，所以目标码基本上是 ASCII，不过也可以通过系统调用 ioctl( )“下载”其他码表。不过，从键码向目标码的转换并不是一个数组就可完成的，因为按下同一个键在不同的情况下应产生出不同的代码。例如，单独按下“A”键时应产生“A”的代码 0x41，而若同时按下 Shift 键(更确切地说是在此之前按下 Shift 键，并且尚未放开)就应产

生“a”的代码 0x61。除此之外，还得考虑 Ctrl 键等其他辅助键，所以实际上需要好几个这样的数组。那么，有几个辅助键影响着目标码的产生呢？一般而言有 4 个，即 Alt 键、Ctrl 键、Altgr 键(右边的 Alt 键)以及 Shift 键。这些辅助键还可以按一定的规则组合，一起来影响目标代码的产生。内核代码中有个全局量 shift_state，以位图的形式记录当前处于按下状态的辅助键，其高 28 位均为 0，低 4 位则用于 4 个辅助键，依次为 Alt、Ctrl、Altgr、Shift。这样，理论上一共可以有 16 个不同的码表，不过有些组合实际上并不使用，所以实际使用的共有 7 个码表。为此，drivers/char/defkeymap.c 中定义了一个指针数组 key_maps[ ]，以 shift_state 为下标就可以找到当前适用的码表。

```
141     ushort *key_maps[MAX_NR_KEYMAPS] = {
142         plain_map, shift_map, altgr_map, 0,
143         ctrl_map, shift_ctrl_map, 0, 0,
144         alt_map, 0, 0, 0,
145         ctrl_alt_map,    0
146     };
```

例如，当没有按下任何辅助键时 shift_state 为 0，所以适用的码表为 plain_map[ ]；同时按下 Shift 键时 shift_state 为 1，所以适用的码表为 shift_map[ ]；同时按下 Alt 键和 Ctrl 键(如 Alt-Ctrl-B) 时 shift_state 为 12，所以适用的码表为 ctrl_alt_map[ ]。具体的码表都在 drivers/char/defkeymap.c 中，这个文件是由工具生成的，编译内核代码时会根据一个码表描述文件 drivers/char/defkeymap.map 自动生成。系统运行时也可以通过系统调用 ioctl( )下载、替换这些码表。系统中有个工具/usr/bin/loadkeys，可以在运行时生成码表并且下载，达到动态地改变码表(从而改变目标语言)的目的。

此外，有些特殊的键盘上有所谓“sticky”辅助键，按一下就将键盘锁定在某种辅助键状态(kbd->slockstate 变成 1)，就好像老是按下 Shift 键不放一样，再按一下就又回到正常状态。有的键盘上还可以将辅助键的作用反转(kbd->lockstate 为 1)。所以，上面的 290～291 行根据当前的 shift_state 和 kbd->slockstate 及 kbd->lockstate 计算出一个下标 shift_final，然后用这个下标在 key_maps[ ]中找到适用的码表(292 行)。接着，如果相应的码表存在，就可以进一步以具体的键码为下标从码表中读出目标代码(295 行)。

由于篇幅的关系，我们不能在这里列出所有 7 个码表，而只列出第一个码表 plain_map[ ]，让读者有个具体的印象(drivers/char/defkeymap.c)：

```
8     u_short plain_map[NR_KEYS] = {
9         0xf200, 0xf01b, 0xf031, 0xf032, 0xf033, 0xf034, 0xf035, 0xf036,
10        0xf037, 0xf038, 0xf039, 0xf030, 0xf02d, 0xf03d, 0xf07f, 0xf009,
11        0xfb71, 0xfb77, 0xfb65, 0xfb72, 0xfb74, 0xfb79, 0xfb75, 0xfb69,
12        0xfb6f, 0xfb70, 0xf05b, 0xf05d, 0xf201, 0xf702, 0xfb61, 0xfb73,
13        0xfb64, 0xfb66, 0xfb67, 0xfb68, 0xfb6a, 0xfb6b, 0xfb6c, 0xf03b,
14        0xf027, 0xf060, 0xf700, 0xf05c, 0xfb7a, 0xfb78, 0xfb63, 0xfb76,
15        0xfb62, 0xfb6e, 0xfb6d, 0xf02c, 0xf02e, 0xf02f, 0xf700, 0xf30c,
16        0xf703, 0xf020, 0xf207, 0xf100, 0xf101, 0xf102, 0xf103, 0xf104,
17        0xf105, 0xf106, 0xf107, 0xf108, 0xf109, 0xf208, 0xf209, 0xf307,
18        0xf308, 0xf309, 0xf30b, 0xf304, 0xf305, 0xf306, 0xf30a, 0xf301,
19        0xf302, 0xf303, 0xf300, 0xf310, 0xf206, 0xf200, 0xf03c, 0xf10a,
```

```
20          0xf10b, 0xf200, 0xf200, 0xf200, 0xf200, 0xf200, 0xf200, 0xf200,
21          0xf30e, 0xf702, 0xf30d, 0xf01c, 0xf701, 0xf205, 0xf114, 0xf603,
22          0xf118, 0xf601, 0xf602, 0xf117, 0xf600, 0xf119, 0xf115, 0xf116,
23          0xf11a, 0xf10c, 0xf10d, 0xf11b, 0xf11c, 0xf110, 0xf311, 0xf11d,
24          0xf200, 0xf200, 0xf200, 0xf200, 0xf200, 0xf200, 0xf200, 0xf200,
25      };
```

例如，“A”键的扫描码为 0x1e，因而键码也是 0x1e（因为小于 89），以此为下标从这个数组中读得其代码为 0xfb61。代码的高 8 位代表类型，低 8 位即为字符“a”的代码 0x61。当高 8 位的数值大于等于 0xf0 时，其数值与 0xf0 之差表示键的类型，小于 0xf0 时则表示采用 Unicode 的 UTF-8 编码。

我们先看键的类型，include/linux/keyboard.h 中共定义了 13 种类型：

```
35      #define KT_LATIN    0   /* we depend on this being zero */
36      #define KT_LETTER   11  /* symbol that can be acted upon by CapsLock */
37      #define KT_FN       1
38      #define KT_SPEC     2
39      #define KT_PAD      3
40      #define KT_DEAD     4
41      #define KT_CONS     5
42      #define KT_CUR      6
43      #define KT_SHIFT    7
44      #define KT_META     8
45      #define KT_ASCII    9
46      #define KT_LOCK     10
47      #define KT_SLOCK    12
```

这里 KT_LATIN 表示普通可打印字符，KT_LETTER 表示字母，KT_FN 为功能键，KT_SPEC 为特殊键，KT_PAD 为“副键盘”或“数字键盘”上的键，KT_SHIFT 表示辅助键，等等。注意这里 KT_ASCII 并不表示“ASCII 码”，而是表示用来输入十六进制数字。还有，KT_DEAD 现在已经不用了，只是为兼容而还保留着。上面“A”键的类型码为 0xfb，所以是 KT_LETTER。

回到 handle_scancode( )的代码中，宏操作 KTYP( )从目标码中取出其高 8 位类型码。如果大于等于 0xf0 即为普通字符(299～312 行)。对于原始模式，输入的扫描码或键码已放入接收队列，但是对某些键的按下或放开也还需要一些附加操作，例如 Shift 键的状态就需要反映在 shift_state 中。这里的常数 TYPES_ALLOWED_IN_RAW_MODE 定义于 drivers/char/keyboard.c：

```
121     /* Key types processed even in raw modes */
122
123     #define TYPES_ALLOWED_IN_RAW_MODE ((1 << KT_SPEC) | (1 << KT_SHIFT))
```

这表示：即使是在原始模式下，对 KT_SPEC 和 KT_SHIFT 这两类键也需要进行一些处理，而其他的就不需要了。

至于正常的键盘操作，那就全都要经过进一步的处理了。其中有些输入要在经过处理以后放入接收队列，有的(如 Shift 键)则处理以后将其丢弃。首先，字母类(KT_LETTER)的输入与普通可打印字符(KT_LATIN)的区别仅在于 CapsLock 键的作用，所以这里把 KT_LETTER 替换成 KT_LATIN，并且根

据当前键盘上的 CapsLock 发光二极管(Led)是否亮着决定是否将 Shift 键的作用反相。

然后，就要根据键的类型调用相应的处理程序了。这里的 key_handler[ ]是个以类型为下标的函数指针数组，定义于 drivers/char/keyboard.c：

```
115     static k_hand key_handler[16] = {
116         do_self, do_fn, do_spec, do_pad, do_dead, do_cons, do_cur, do_shift,
117         do_meta, do_ascii, do_lock, do_lowercase, do_slock, do_dead2,
118         do_ignore, do_ignore
119     };
```

数组的大小是 16，但是前面实际上只定义了 13 种类型，所以最后 3 个函数指针是不可能用到的。我们先看看对 KT_LATIN，即普通可打印字符的处理，函数 do_self( )的代码在 drivers/char/keyboard.c 中：

```
542     static void do_self(unsigned char value, char up_flag)
543     {
544         if (up_flag)
545             return;      /* no action, if this is a key release */
546
547         if (diacr)
548             value = handle_diacr(value);
549
550         if (dead_key_next) {
551             dead_key_next = 0;
552             diacr = value;
553             return;
554         }
555
556         put_queue(value);
557     }
```

首先，up_flag 非 0 表示当前的事件是放开(而不是按下)一个键，所以什么事也不用干，把输入丢弃就是了。否则，如果全局量 diacr 和 dead_key_next 都是 0，那就是一个正常的输入字符了，所以通过 put_queue( )把这个字符放在接收队列中。

那么，diacr 和 dead_key_next 是干什么用的呢？原来，在许多拼音文字中都有所谓“accent”字符(也称为“diacritical”字符)，例如“A”顶上加个小圈就是一个。输入这个特殊字符时先要同时按一下 Ctrl 和“.” 键，然后按“o”键，最后按“A” 键。由于这是个序列，所以又得要实现一个“有限状态机”，而 diacr 和 dead_key_next 就是用于这个目的。

以刚才讲的序列为例，“.”键的键码为 0x34，但是因为同时按 Ctrl 键，所以从 ctrl_map[ ]中(这里没有列出)读出目标码为 0xf20e，其类型为 KT_SPEC。于是，从 key_handler[ ]中找到函数指针为 do_spec( )，其代码也在 drivers/char/keyboard.c 中：

```
525     static void do_spec(unsigned char value, char up_flag)
526     {
```

```
527         if (up_flag)
528             return;
529         if (value >= SIZE(spec_fn_table))
530             return;
531         if ((kbd->kbdmode == VC_RAW || kbd->kbdmode == VC_MEDIUMRAW) &&
532             !(SPECIALS_ALLOWED_IN_RAW_MODE & (1 << value)))
533             return;
534         spec_fn_table[value]( );
535     }
```

长话短说，这里最后又通过同一文件(keyboard.c)中的函数指针数组 spec_fn_table[ ]执行一个函数：

```
132     static void fnp spec_fn_table[ ] = {
133         do_null,      enter,       show_ptregs,      show_mem,
134         show_state,  send_intr,   lastcons,     caps_toggle,
135         num,          hold,        scroll_forw,      scroll_back,
136         boot_it,      caps_on,     compose,      SAK,
137         decr_console,   incr_console,    spawn_console,   bare_num
138     };
```

由于目标码中的低 8 位为 0x0e，所以执行的是 compose( )，还是在同一文件中：

```
488     static void compose(void)
489     {
490         dead_key_next = 1;
491     }
```

这样，当输入“o”键时，在 do_self( )中就因 dead_key_next 为 1 而将该字符的代码存放在 diacr 中，dead_key_next 则又清成 0。接着，当输入“A”键时则因 diacr 非 0 而调用 handle_diacr( )，其代码仍在 drivers/char/keyboard.c 中：

```
589     /*
590      * We have a combining character DIACR here, followed by the character CH.
591      * If the combination occurs in the table, return the corresponding value.
592      * Otherwise, if CH is a space or equals DIACR, return DIACR.
593      * Otherwise, conclude that DIACR was not combining after all,
594      * queue it and return CH.
595      */
596     unsigned char handle_diacr(unsigned char ch)
597     {
598         int d = diacr;
599         int i;
600
601         diacr = 0;
602
603         for (i = 0; i < accent_table_size; i++) {
```

```
604             if (accent_table[i].diacr == d && accent_table[i].base == ch)
605                 return accent_table[i].result;
606         }
607
608         if (ch == ' ' || ch == d)
609             return d;
610
611         put_queue(d);
612         return ch;
613     }
```

每个这样的合成字符都有一个数据结构，根据前后两个字符给出合成字符的代码，这些数据结构都在一个数组 accent_table[ ]中，所以这里搜索这个数组并返回结果(605 行)。返回的合成字符则在 do_self( )中放入接收队列。

我们在这里当然不可能逐一去读 key_handler[ ]中那些函数的代码，读者可以自己阅读。

回到 handle_scancode( )的代码中，我们已经看了类型码大于等于 0xf0 时的处理。前面我们也提到过，如果类型码小于 0xf0 就表示采用 Unicode 的 UTF-8 编码。为方便阅读，我们再把 handle_scancode( )中有关的几行列出于下：

```
313                 } else {
314                     /* maybe only if (kbd->kbdmode == VC_UNICODE) ? */
315                     if (!up_flag && !raw_mode)
316                       to_utf8(keysym);
317                 }
```

这里执行 to_utf8( )的条件是不言自明的，to_utf8( )的代码在 drivers/char/keyboard.c 中：

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_keyboard_event( ) > handle_scancode( ) > to_utf8( )]

```
171     void to_utf8(ushort c) {
172         if (c < 0x80)
173         put_queue(c);            /*  0*******  */
174         else if (c < 0x800) {
175         put_queue(0xc0 | (c >> 6));      /*  110***** 10******  */
176         put_queue(0x80 | (c & 0x3f));
177         } else {
178         put_queue(0xe0 | (c >> 12));     /*  1110**** 10****** 10******  */
179         put_queue(0x80 | ((c >> 6) & 0x3f));
180         put_queue(0x80 | (c & 0x3f));
181         }
182         /* UTF-8 is defined for words of up to 31 bits,
183            but we need only 16 bits here */
184     }
```

这段代码正好可以让读者复习和消化前一节中讲到的 UTF-8 编码。注意这里没有对各种键作分类处理，而只是把转换成的代码依次放入接收队列，在这方面有点像原始模式。此外，这里的 Unicode

仍是一键一码，所以这些程序只能用于拼音文字。

再回到handle_scancode( )的代码中，最后还有一种可能，就是key_maps[ ]中不存在由当前shift_final决定的码表(318～330 行)。既然没有码表，那就只好把输入丢弃。不过，前面已经根据最新的输入改变了当前的位图 key_down[ ]，也许最新的输入恰好是辅助键的状态变化，因而也应该反映在位图shift_state中，所以要根据位图key_down[ ]的内容计算出最新的shift_state。我们再列出handle_scancode( )中有关的几行如下：

```
322                compute_shiftstate( );
323                kbd->slockstate = 0; /* play it safe */
```

这里 compute_shiftstate( )的代码在 drivers/char/keyboard.c 中：

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_keyboard_event( ) > handle_scancode( ) > compute_shiftstate( )]

```
737    /* called after returning from RAW mode or when changing consoles -
738       recompute k_down[ ] and shift_state from key_down[ ] */
739    /* maybe called when keymap is undefined, so that shiftkey release is seen */
740    void compute_shiftstate(void)
741    {
742        int i, j, k, sym, val;
743
744        shift_state = 0;
745        for(i=0; i < SIZE(k_down); i++)
746          k_down[i] = 0;
747
748        for(i=0; i < SIZE(key_down); i++)
749          if(key_down[i]) { /* skip this word if not a single bit on */
750            k = i*BITS_PER_LONG;
751            for(j=0; j<BITS_PER_LONG; j++,k++)
752              if(test_bit(k, key_down)) {
753            sym = U(plain_map[k]);
754            if(KTYP(sym) == KT_SHIFT || KTYP(sym) == KT_SLOCK) {
755              val = KVAL(sym);
756              if (val == KVAL(K_CAPSSHIFT))
757                val = KVAL(K_SHIFT);
758              k_down[val]++;
759              shift_state |= (1<<val);
760            }
761              }
762          }
763    }
```

这里依次扫描位图 key_down[ ]中的每一位，对当前处于按下状态的每一个键都从 plain_map[ ]中读出其代码，看看是什么类型。如果是辅助键或锁定键则将 shift_state 中的相应位设成 1。

完成了 handle_scancode( )的执行以后，对键盘中断的服务也就基本上完成了。

总而言之，键盘中断是由键盘上的事件引起的，根据键盘运行模式的不同，有些事件导致字符的接收，有的则只引起键盘驱动程序底层的某些状态变化。对于导致字符接收的事件，所接收的代码视键盘运行模式的不同和适用码表的不同而异，分别可以是扫描码、键码或者目标码。这些代码通常只是一个字节，但是在有些情况下也可能是个字节序列。然而，有一点是共同的，凡是接收的代码都通过 put_queue( )逐个字节地放入“前台”键盘的接收队列。这个函数的代码在 drivers/char/keyboard.c 中：

```
335    void put_queue(int ch)
336    {
337        wake_up(&keypress_wait);
338        if (tty) {
339            tty_insert_flip_char(tty, ch, 0);
340            con_schedule_flip(tty);
341        }
342    }
```

对于从键盘接收到而未经进一步加工的字符，先放在tty_struct结构内的一个所谓“flip缓冲区”中(include/linux/tty_flip.h)：

[put_queue( ) > tty_insert_flip_char( )]

```
10     _INLINE_ void tty_insert_flip_char(struct tty_struct *tty,
11                         unsigned char ch, char flag)
12     {
13         if (tty->flip.count < TTY_FLIPBUF_SIZE) {
14             tty->flip.count++;
15             *tty->flip.flag_buf_ptr++ = flag;
16             *tty->flip.char_buf_ptr++ = ch;
17         }
18     }
```

在中断服务的过程中，CPU 对从键盘读入的数据进行了一些处理和转换。相对而言，这些处理和转换都是很简单的，可以在很短的时间内完成。可是，对于产生输入字符的事件来说，已经进行的这些处理还只是“万里长征走完了第一步”，接下去还要进行不少处理，而且那些处理就比较费时间了。我们在第 3 章中讲到过，对于这种本应在中断服务程序中进行，可是又比较费时间的操作应该放在 bh 函数中或者作为 tasklet 在比较宽松(允许中断)的条件下执行。所以，将字符放入 flip 缓冲区以后，还要通过 con_schedule_flip( )调度控制台终端的 tasklet 运行(include/linux/kbd_kern.h)。控制台终端的 tasklet 是 console_tasklet( )。所谓调度其运行，就是将一个指向这个函数的指针通过一个结构挂入 tasklet 的执行队列。

[put_queue( ) > con_schedule_flip( )]

```
164    extern inline void con_schedule_flip(struct tty_struct *t)
165    {
166        queue_task(&t->flip.tqueue, &con_task_queue);
167        tasklet_schedule(&console_tasklet);
```

```
168     }
```

这个 tasklet 定义于 drivers/char/console.c 中：

```
2372    DECLARE_TASKLET_DISABLED(console_tasklet, console_softint, 0);
```

读者应结合第 3 章中的有关内容搞清其机理。这个 tasklet 的执行程序是 console_softint( )，其代码也在 drivers/char/console.c 中：

```
2003    /*
2004     * This is the console switching tasklet.
2005     *
2006     * Doing console switching in a tasklet allows
2007     * us to do the switches asynchronously (needed when we want
2008     * to switch due to a keyboard interrupt).  Synchronization
2009     * with other console code and prevention of re-entrancy is
2010     * ensured with console_lock.
2011     */
2012    static void console_softint(unsigned long ignored)
2013    {
2014        /* Runs the task queue outside of the console lock.  These
2015         * callbacks can come back into the console code and thus
2016         * will perform their own locking.
2017         */
2018        run_task_queue(&con_task_queue);
2019
2020        spin_lock_irq(&console_lock);
2021
2022        if (want_console >= 0) {
2023            if (want_console != fg_console && vc_cons_allocated(want_console)) {
2024                hide_cursor(fg_console);
2025                change_console(want_console);
2026                /* we only changed when the console had already
2027                   been allocated - a new console is not created
2028                   in an interrupt routine */
2029            }
2030            want_console = -1;
2031        }
2032        if (do_poke_blanked_console) { /* do not unblank for a LED change */
2033            do_poke_blanked_console = 0;
2034            poke_blanked_console( );
2035        }
2036        if (scrollback_delta) {
2037            int currcons = fg_console;
2038            clear_selection( );
2039            if (vcmode == KD_TEXT)
2040                sw->con_scrolldelta(vc_cons[currcons].d, scrollback_delta);
```

```
2041            scrollback_delta = 0;
2042        }
2043
2044        spin_unlock_irq(&console_lock);
2045    }
```

进入 console_softint( )，我们就上了一个层次，从键盘上升到了终端设备的层次。

使用者通过按 Alt 键和某个功能键(例如 F2)切换虚拟终端时，具体的切换也是在这里完成的，不过我们现在对此不感兴趣。我们在这里关心的是对 run_task_queue( )的调用。这个队列中有些什么函数要执行呢？前面我们看到，con_schedule_flip( )中把相应 tty_struct 结构中的 flip.tqueue 挂入了这个队列，而在前一节中打开文件时调用的 initialize_tty_struct( )里面，则将其函数指针设置成指向 flush_to_ldisc( )，这个函数的代码在 drivers/char/tty_io.c 中：

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( )]

```
1863    /*
1864     * This routine is called out of the software interrupt to flush data
1865     * from the flip buffer to the line discipline.
1866     */
1867    static void flush_to_ldisc(void *private_)
1868    {
1869        struct tty_struct *tty = (struct tty_struct *) private_;
1870        unsigned char   *cp;
1871        char        *fp;
1872        int     count;
1873        unsigned long flags;
1874
1875        if (test_bit(TTY_DONT_FLIP, &tty->flags)) {
1876            queue_task(&tty->flip.tqueue, &tq_timer);
1877            return;
1878        }
1879        if (tty->flip.buf_num) {
1880            cp = tty->flip.char_buf + TTY_FLIPBUF_SIZE;
1881            fp = tty->flip.flag_buf + TTY_FLIPBUF_SIZE;
1882            tty->flip.buf_num = 0;
1883
1884            save_flags(flags); cli();
1885            tty->flip.char_buf_ptr = tty->flip.char_buf;
1886            tty->flip.flag_buf_ptr = tty->flip.flag_buf;
1887        } else {
1888            cp = tty->flip.char_buf;
1889            fp = tty->flip.flag_buf;
1890            tty->flip.buf_num = 1;
1891
1892            save_flags(flags); cli();
1893            tty->flip.char_buf_ptr = tty->flip.char_buf + TTY_FLIPBUF_SIZE;
```

```
1894            tty->flip.flag_buf_ptr = tty->flip.flag_buf + TTY_FLIPBUF_SIZE;
1895        }
1896        count = tty->flip.count;
1897        tty->flip.count = 0;
1898        restore_flags(flags);
1899
1900        tty->ldisc.receive_buf(tty, cp, fp, count);
1901    }
```

这个函数的任务是通过 tty->ldisc.receive_buf( )从 flip 缓冲区中把数据搬运到另一个缓冲区中，同时加以处理。

可是，这里隐含着一个问题。我们知道，tasklet 在执行过程中是允许中断的(否则就没有意义了)，如果 flush_to_ldisc( )正在从一个缓冲区往外读数据，而正好又发生了键盘中断，从而又通过 put_queue( )往同一缓冲区里面写数据，那岂不是乱了套？可是，我们所熟知的一些保证互斥的办法在这里却都用不上。首先，tasklet 和中断服务程序都不是在某个进程的上下文中，因而不能睡眠。其次。加锁也行不通，因为这二者可能(在单 CPU 的系统中则肯定)由同一个 CPU 执行。当然，可以在执行的过程中关闭中断以免打扰，可是那样又违背了设置和使用 tasklet 机制的初衷。这里的解决办法是采用“双缓冲”，即交替使用两个缓冲区。这就好像翻来覆去使用一张纸的两面一样，我在用这一面就让你用另一面，等我用完这一面就再翻个面。正因为这样，才称之为“flip”缓冲区，其数据结构定义于 include/linux/tty.h：

```
139    struct tty_flip_buffer {
140        struct tq_struct    tqueue;
141        struct semaphore    pty_sem;
142        char                *char_buf_ptr;
143        unsigned char       *flag_buf_ptr;
144        int                 count;
145        int                 buf_num;
146        unsigned char       char_buf[2*TTY_FLIPBUF_SIZE];
147        char                flag_buf[2*TTY_FLIPBUF_SIZE];
148        unsigned char       slop[4];    /* N.B. bug overwrites buffer by 1 */
149    };
```

可见，缓冲区 char_buf[ ]的大小是 TTY_FLIPBUF_SIZE 的两倍，这就好像一张纸的两面。与 char_buf[ ]相平行还有一个数组 flag_buf[ ]，一个字符总是与一个 flag 字节配对的，不过从 put_queue( )调用 tty_insert_flip_char( )时总是把 flag 字节设成 0。指针 char_buf_ptr 和 flag_buf_ptr 都是给 tty_insert_flip_char( )使用的，而 flush_to_ldisc( )中对这些指针的设置就起着把纸翻一个面的作用。当然，这种方法也是有缺点的，那就是实际上把缓冲区的容量减小了一半；而 tty_insert_flip_char( )在发现缓冲区已满的情况下就会把接收到的字符丢弃。所以，在 flush_to_ldisc( )中保留了一种选择，如果把 tty_struct 结构中的标志位 TTY_DONT_FLIP 设成 1，那就把要执行的函数转到 tq_timer 队列中去，让它在时钟中断时在关中断的条件下执行。在前面 read_chan( )的代码中，我们看到当高层从缓冲区读出时就把 TTY_DONT_FLIP 标志位设成 1，进入睡眠时便把它改为 0。

具体的操作取决于相应 tty_ldisc 数据结构中的函数指针 receive_buf。对于 tty_ldisc_N_TTY 而言，这个指针指向 n_tty_receive_buf( )，其代码在 drivers/char/n_tty.c 中：

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( ) > n_tty_receive_buf( )]

```
705     static void n_tty_receive_buf(struct tty_struct *tty, const unsigned char *cp,
706                         char *fp, int count)
707     {
708         const unsigned char *p;
709         char *f, flags = TTY_NORMAL;
710         int i;
711         char    buf[64];
712         unsigned long cpuflags;
713 
714         if (!tty->read_buf)
715             return;
716 
717         if (tty->real_raw) {
718             spin_lock_irqsave(&tty->read_lock, cpuflags);
719             i = MIN(count, MIN(N_TTY_BUF_SIZE - tty->read_cnt,
720                         N_TTY_BUF_SIZE - tty->read_head));
721             memcpy(tty->read_buf + tty->read_head, cp, i);
722             tty->read_head = (tty->read_head + i) & (N_TTY_BUF_SIZE-1);
723             tty->read_cnt += i;
724             cp += i;
725             count -= i;
726 
727             i = MIN(count, MIN(N_TTY_BUF_SIZE - tty->read_cnt,
728                         N_TTY_BUF_SIZE - tty->read_head));
729             memcpy(tty->read_buf + tty->read_head, cp, i);
730             tty->read_head = (tty->read_head + i) & (N_TTY_BUF_SIZE-1);
731             tty->read_cnt += i;
732             spin_unlock_irqrestore(&tty->read_lock, cpuflags);
733         } else {
734             for (i=count, p = cp, f = fp; i; i--, p++) {
735                 if (f)
736                     flags = *f++;
737                 switch (flags) {
738                 case TTY_NORMAL:
739                     n_tty_receive_char(tty, *p);
740                     break;
741                 case TTY_BREAK:
742                     n_tty_receive_break(tty);
743                     break;
744                 case TTY_PARITY:
745                 case TTY_FRAME:
746                     n_tty_receive_parity_error(tty, *p);
747                     break;
748                 case TTY_OVERRUN:
749                     n_tty_receive_overrun(tty);
750                     break;
```

```
751                 default:
752                     printk("%s: unknown flag %d\n",
753                            tty_name(tty, buf), flags);
754                     break;
755                 }
756             }
757             if (tty->driver.flush_chars)
758                 tty->driver.flush_chars(tty);
759         }
760 
761         if (!tty->icanon && (tty->read_cnt >= tty->minimum_to_wake)) {
762             kill_fasync(&tty->fasync, SIGIO, POLL_IN);
763             if (waitqueue_active(&tty->read_wait))
764                 wake_up_interruptible(&tty->read_wait);
765         }
766 
767         /*
768          * Check the remaining room for the input canonicalization
769          * mode.  We don't want to throttle the driver if we're in
770          * canonical mode and don't have a newline yet!
771          */
772         if (n_tty_receive_room(tty) < TTY_THRESHOLD_THROTTLE) {
773             /* check TTY_THROTTLED first so it indicates our state */
774             if (!test_and_set_bit(TTY_THROTTLED, &tty->flags) &&
775                 tty->driver.throttle)
776                 tty->driver.throttle(tty);
777         }
778     }
```

终端的 tty_struct 结构中有 raw 和 real_raw 两个字段，都表示终端层次上的原始模式，但是 real_raw 更为原始，表示即使在链路上出现了 Break 状态，或者接收到的数据有奇偶校验出错，也都原封不动上交。具体的设置取决于所用的 termio，但可以通过系统调用 ioctl( )加以改变(命令码为 TCSETSF)。不过，对于控制台终端这二者实际上没有多大区别。

可想而知，对于运行于 real_raw 模式的终端，只要把 flip 缓冲区中的内容复制到终端的 read_buf[ ] 缓冲区中就可以了(718～732 行)，否则就要通过 734～756 行的 for 循环逐个字节地边搬运边处理加工，这就是比较费时间的操作所在。

如前所述，flip 缓冲区中的代码字节都是与一个 flag 字节配对存在的，这个 flag 字节指明了代码字节的性质和类型。不过，由 tty_insert_flip_char( )写入 flip 缓冲区时总是把相应的 flag 字节设成 0，就是这里的 TTY_NORMAL，所以总是调用 n_tty_receive_char( )，其代码也在 drivers/char/n_tty.c 中。这个函数比较长，我们需要分段阅读。

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( ) > n_tty_receive_buf( ) > n_tty_receive_char( )]

```
500     static inline void n_tty_receive_char(struct tty_struct *tty, unsigned char c)
501     {
```

```
502         if (tty->raw) {
503             put_tty_queue(c, tty);
504             return;
505         }
506 
507         if (tty->stopped && !tty->flow_stopped &&
508             I_IXON(tty) && I_IXANY(tty)) {
509             start_tty(tty);
510             return;
511         }
512 
513         if (I_ISTRIP(tty))
514             c &= 0x7f;
515         if (I_IUCLC(tty) && L_IEXTEN(tty))
516             c=tolower(c);
517 
518         if (tty->closing) {
519             if (I_IXON(tty)) {
520                 if (c == START_CHAR(tty))
521                     start_tty(tty);
522                 else if (c == STOP_CHAR(tty))
523                     stop_tty(tty);
524             }
525             return;
526         }
527 
528         /*
529          * If the previous character was LNEXT, or we know that this
530          * character is not one of the characters that we'll have to
531          * handle specially, do shortcut processing to speed things
532          * up.
533          */
534         if (!test_bit(c, &tty->process_char_map) || tty->lnext) {
535             finish_erasing(tty);
536             tty->lnext = 0;
537             if (L_ECHO(tty)) {
538                 if (tty->read_cnt >= N_TTY_BUF_SIZE-1) {
539                     put_char('\a', tty); /* beep if no space */
540                     return;
541                 }
542                 /* Record the column of first canon char. */
543                 if (tty->canon_head == tty->read_head)
544                     tty->canon_column = tty->column;
545                 echo_char(c, tty);
546             }
547             if (I_PARMRK(tty) && c == (unsigned char) '\377')
548                 put_tty_queue(c, tty);
549             put_tty_queue(c, tty);
```

```
550             return;
551         }
552
```

首先，如果终端运行于原始模式，就只是简单地将从 flip 缓冲区读出的字符写入终端的 read_buf[ ]缓冲区。

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( ) > n_tty_receive_buf( ) > n_tty_receive_char( ) > put_tty_queue( )]

```
 89     static inline void put_tty_queue(unsigned char c, struct tty_struct *tty)
 90     {
 91         unsigned long flags;
 92         /*
 93          *  The problem of stomping on the buffers ends here.
 94          *  Why didn't anyone see this one comming? --AJK
 95         */
 96         spin_lock_irqsave(&tty->read_lock, flags);
 97         if (tty->read_cnt < N_TTY_BUF_SIZE) {
 98             tty->read_buf[tty->read_head] = c;
 99             tty->read_head = (tty->read_head + 1) & (N_TTY_BUF_SIZE-1);
100             tty->read_cnt++;
101         }
102         spin_unlock_irqrestore(&tty->read_lock, flags);
103     }
```

终端的 read_buf[ ]缓冲区是个环形缓冲区，tty->read_head 总是指向下一个可以写入的位置。如果 read_buf[ ]中已经写满，则丢弃新来到的输入。

对于非原始模式，或称"加工模式"，则要进行一系列的处理。这些处理包括一般的和特殊的两部分。前者包括终端的自动暂停(进入省电模式)和启动、强制转换成小写字符、往显示屏幕上"回打"等。后者则包括许多因具体字符而异的处理，其中也包括基于 XON/XOFF 字符(一般是 Ctrl-Q 和 Ctrk-S)的流量控制。

不过，真正要加以特殊处理的字符毕竟还是少数，tty_struct 结构中的位图 process_char_map 指明了需要特殊处理的字符。只要一个字符不属于这个位图所代表的集合，对其进一步的处理就只限于"回打"，然后就通过 put_tty_queue( )把该字符写入 read_buf[ ]缓冲区(535~550 行)。如果此前的字符是个"erase"字符(如键盘上的 Backspace 键)，则终端处于正在退字符的状态，现在新的字符既然不属于 process_char_map，就显然不再是"erase"字符，所以先调用 finish_erasing( )结束终端的退字符状态，然后就是对"回打"的处理了。

如果终端的模式设置表明需要回打，就调用 echo_char( )完成这个操作。但是先要检查一下 read_buf[ ]缓冲区中是否还有空位可以接受当前的字符。如果已经满了就要向显示器接口写一个"\a"字符，让它"嘟"地响一下，以免使用者误以为打入的字符已被接受。注意不要把这里调用的 put_char( )与应用程序设计中的 C 库函数混淆，这是个 inline 函数，定义于 drivers/char/n_tty.c:

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( ) > n_tty_receive_buf( ) > n_tty_receive_char( ) >

put_char( )]

```
301     static inline void put_char(unsigned char c, struct tty_struct *tty)
302     {
303         tty->driver.put_char(tty, c);
304     }
```

显然，具体的操作取决于相应 tty_driver 结构中的函数指针 put_char，对于一般的终端设备这个指针指向 tty_default_put_char( )，等一下我们还要看这个函数的代码。

函数 echo_char( )的代码在 drivers/char/n_tty.c 中：

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( ) > n_tty_receive_buf( )
 > n_tty_receive_char( ) > echo_char( )]

```
308     static void echo_char(unsigned char c, struct tty_struct *tty)
309     {
310         if (L_ECHOCTL(tty) && iscntrl(c) && c != '\t') {
311             put_char('^', tty);
312             put_char(c ^ 0100, tty);
313             tty->column += 2;
314         } else
315             opost(c, tty);
316     }
```

对于输入时同时按下了 Ctrl 键的字符，回打时要在该字符之前加上一个“^”字符。对正常输入的字符则还需要进一步的特殊处理，情况更为复杂，所以通过 opost( )完成回打，其代码也在 drivers/char/n_tty.c 中：

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( ) > n_tty_receive_buf( )
 > n_tty_receive_char( ) > echo_char( ) > opost( )]

```
172     /*
173      * Perform OPOST processing.  Returns -1 when the output device is
174      * full and the character must be retried.
175      */
176     static int opost(unsigned char c, struct tty_struct *tty)
177     {
178         int space, spaces;
179
180         space = tty->driver.write_room(tty);
181         if (!space)
182             return -1;
183
184         if (O_OPOST(tty)) {
185             switch (c) {
186             case '\n':
187                 if (O_ONLRET(tty))
```

```
188                 tty->column = 0;
189             if (O_ONLCR(tty)) {
190                 if (space < 2)
191                     return -1;
192                 tty->driver.put_char(tty, '\r');
193                 tty->column = 0;
194             }
195             tty->canon_column = tty->column;
196             break;
197         case '\r':
198             if (O_ONOCR(tty) && tty->column == 0)
199                 return 0;
200             if (O_OCRNL(tty)) {
201                 c = '\n';
202                 if (O_ONLRET(tty))
203                     tty->canon_column = tty->column = 0;
204                 break;
205             }
206             tty->canon_column = tty->column = 0;
207             break;
208         case '\t':
209             spaces = 8 - (tty->column & 7);
210             if (O_TABDLY(tty) == XTABS) {
211                 if (space < spaces)
212                     return -1;
213                 tty->column += spaces;
214                 tty->driver.write(tty, 0, "        ", spaces);
215                 return 0;
216             }
217             tty->column += spaces;
218             break;
219         case '\b':
220             if (tty->column > 0)
221                 tty->column--;
222             break;
223         default:
224             if (O_OLCUC(tty))
225                 c = toupper(c);
226             if (!iscntrl(c))
227                 tty->column++;
228             break;
229         }
230     }
231     tty->driver.put_char(tty, c);
232     return 0;
233 }
```

首先检查终端的写缓冲区中是否有空间。这里的特殊处理主要是针对“\r”、“\n”、“\t”以及“\b”

进行的。对于“换行”键“\n”的考虑主要是自动在前面插入一个“回车”字符“\r”；对“\t”的处理则是要把它展开成若干个空格。这段程序比较简单，我们就不多说了。最后，与 put_char() 一样，也是通过相应 tty_driver 结构中的函数指针 put_char 完成对终端显示器的写操作。对于一般的终端设备，这个指针指向 tty_default_put_char()，其代码还是在 drivers/char/tty_io.c 中：

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( ) > n_tty_receive_buf( )
 > n_tty_receive_char( ) > echo_char( ) > opost( ) > tty_default_put_char( )]

```
1979    /*
1980     * The default put_char routine if the driver did not define one.
1981     */
1982    void tty_default_put_char(struct tty_struct *tty, unsigned char ch)
1983    {
1984        tty->driver.write(tty, 0, &ch, 1);
1985    }
```

这又是通过函数指针的调用。对于控制台终端，这个指针指向 con_write( )，其代码在 drivers/char/console.c 中：

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( ) > n_tty_receive_buf( )
 > n_tty_receive_char( ) > echo_char( ) > opost( ) > tty_default_put_char( ) > con_write( )]

```
2217    static int con_write(struct tty_struct * tty, int from_user,
2218                const unsigned char *buf, int count)
2219    {
2220        int retval;
2221
2222        pm_access(pm_con);
2223        retval = do_con_write(tty, from_user, buf, count);
2224        con_flush_chars(tty);
2225
2226        return retval;
2227    }
```

注意这里的第二个参数 from_user，当这个参数为 1 时表示要写的内容来自用户空间，为 0 时则来自系统空间。指针 buf 指向要写的字符缓冲区，count 则为长度。显然，这个函数是个汇合点，一边是系统调用 read( )中的回打操作，另一边则是常规的系统调用 write( )。

操作的主体是 do_con_write( )，也在 drivers/char/tty_io.c 中。这个函数又比较长，我们分段阅读。

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( ) > n_tty_receive_buf( )
 > n_tty_receive_char( ) > echo_char( ) > opost( ) > tty_default_put_char( ) > con_write( )
 > do_con_write( )]

```
1807    static int do_con_write(struct tty_struct * tty, int from_user,
1808                const unsigned char *buf, int count)
1809    {
```

```
1810    #ifdef VT_BUF_VRAM_ONLY
1811    #define FLUSH do { } while(0);
1812    #else
1813    #define FLUSH if (draw_x >= 0) { \
1814        sw->con_putcs(vc_cons[currcons].d, (u16 *)draw_from, \
                             (u16 *)draw_to-(u16 *)draw_from, y, draw_x); \
1815        draw_x = -1; \
1816        }
1817    #endif
1818
1819        int c, tc, ok, n = 0, draw_x = -1;
1820        unsigned int currcons;
1821        unsigned long draw_from = 0, draw_to = 0;
1822        struct vt_struct *vt = (struct vt_struct *)tty->driver_data;
1823        u16 himask, charmask;
1824        const unsigned char *orig_buf = NULL;
1825        int orig_count;
1826
1827        currcons = vt->vc_num;
1828        if (!vc_cons_allocated(currcons)) {
1829            /* could this happen? */
1830            static int error = 0;
1831            if (!error) {
1832            error = 1;
1833            printk("con_write: tty %d not allocated\n", currcons+1);
1834            }
1835            return 0;
1836        }
1837
1838        orig_buf = buf;
1839        orig_count = count;
1840
1841        if (from_user) {
1842            down(&con_buf_sem);
1843
1844    again:
1845            if (count > CON_BUF_SIZE)
1846                count = CON_BUF_SIZE;
1847            if (copy_from_user(con_buf, buf, count)) {
1848                n = 0; /* ?? are error codes legal here ?? */
1849                goto out;
1850            }
1851
1852            buf = con_buf;
1853        }
1854
```

这一段程序比较简单，主要是从用户空间把输出数据复制到内核中。对于我们这个情景，输出数

据本来就在内核中，所以实际上不起作用。我们继续往下看：

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( ) > n_tty_receive_buf( ) > n_tty_receive_char( ) > echo_char( ) > opost( ) > tty_default_put_char( ) > con_write( ) > do_con_write( )]

```
1855        /* At this point 'buf' is guarenteed to be a kernel buffer
1856         * and therefore no access to userspace (and therefore sleeping)
1857         * will be needed.  The con_buf_sem serializes all tty based
1858         * console rendering and vcs write/read operations.  We hold
1859         * the console spinlock during the entire write.
1860         */
1861
1862        spin_lock_irq(&console_lock);
1863
1864        himask = hi_font_mask;
1865        charmask = himask ? 0x1ff : 0xff;
1866
1867        /* undraw cursor first */
1868        if (IS_FG)
1869            hide_cursor(currcons);
1870
1871        while (!tty->stopped && count) {
1872            c = *buf;
1873            buf++;
1874            n++;
1875            count--;
1876
1877            if (utf) {
1878                /* Combine UTF-8 into Unicode */
1879                /* Incomplete characters silently ignored */
1880                if(c > 0x7f) {
1881                if (utf_count > 0 && (c & 0xc0) == 0x80) {
1882                    utf_char = (utf_char << 6) | (c & 0x3f);
1883                    utf_count--;
1884                    if (utf_count == 0)
1885                        tc = c = utf_char;
1886                    else continue;
1887                } else {
1888                    if ((c & 0xe0) == 0xc0) {
1889                        utf_count = 1;
1890                        utf_char = (c & 0x1f);
1891                    } else if ((c & 0xf0) == 0xe0) {
1892                        utf_count = 2;
1893                        utf_char = (c & 0x0f);
1894                    } else if ((c & 0xf8) == 0xf0) {
1895                        utf_count = 3;
1896                        utf_char = (c & 0x07);
1897                    } else if ((c & 0xfc) == 0xf8) {
```

```
1898                        utf_count = 4;
1899                        utf_char = (c & 0x03);
1900                    } else if ((c & 0xfe) == 0xfc) {
1901                        utf_count = 5;
1902                        utf_char = (c & 0x01);
1903                    } else
1904                        utf_count = 0;
1905                    continue;
1906                      }
1907                } else {
1908                  tc = c;
1909                  utf_count = 0;
1910                }
1911            } else {     /* no utf */
1912              tc = translate[toggle_meta ? (c|0x80) : c];
1913            }
```

先通过 hide_cursor( )把光标隐去，然后就是对输出数据的 while 循环。这里的 utf 是个宏定义，定义于 drivers/char/console_macros.h：

```
26     #define utf       (vc_cons[currcons].d->vc_utf)
27     #define utf_count     (vc_cons[currcons].d->vc_utf_count)
28     #define utf_char      (vc_cons[currcons].d->vc_utf_char)
```

就是说，如果当前(前台)虚拟控制台终端运行于 UTF-8 模式，则要把 UTF-8 代码还原成 16 位的 Unicode，最后存放在变量 tc 中。我们把这段程序留给读者与前面的 to_utf8( )对照阅读，这里仅关注普通的 8 位代码。用于显示的字符统一为 16 位 Unicode，一般的 8 位字符要通过一个数组变换成 Unicode，这里的 translate 在 drivers/char/console_macros.h 中定义成当前终端的变换数组。

```
21     #define translate     (vc_cons[currcons].d->vc_translate)
```

每个虚拟控制台的 vc_data 数据结构中有个指针 vc_translate，指向一个大小为 256 的数组，8 位的 ASCII 或其他代码可以通过这个数组转换成 16 位 Unicode。内核中有个二维数组 translations[ ][ ]，定义于 drivers/char/consolemap.c，我们在这里只列出其中若干片断：

```
24     static unsigned short translations[ ][256] = {
25       /* 8-bit Latin-1 mapped to Unicode -- trivial mapping */
26       {
27         0x0000, 0x0001, 0x0002, 0x0003, 0x0004, 0x0005, 0x0006, 0x0007,
28         0x0008, 0x0009, 0x000a, 0x000b, 0x000c, 0x000d, 0x000e, 0x000f,
           . . . . . . .
59       },
60       /* VT100 graphics mapped to Unicode */
61       {
           . . . . . . .
```

```
94        },
95        /* IBM Codepage 437 mapped to Unicode */
96        {
97          0x0000, 0x263a, 0x263b, 0x2665, 0x2666, 0x2663, 0x2660, 0x2022,
98          0x25d8, 0x25cb, 0x25d9, 0x2642, 0x2640, 0x266a, 0x266b, 0x263c,
          . . . . . . .
129       },
130       /* User mapping -- default to codes for direct font mapping */
131       {
132         0xf000, 0xf001, 0xf002, 0xf003, 0xf004, 0xf005, 0xf006, 0xf007,
133         0xf008, 0xf009, 0xf00a, 0xf00b, 0xf00c, 0xf00d, 0xf00e, 0xf00f,
          . . . . . . .
164       }
165     };
```

这个二维数组中包含着 4 个字符转换数组，每个虚拟控制台的指针 vc_translate 指向其中的一个，可以通过系统调用 ioctl( )来改变。不仅如此，还可以通过 ioctl( )从用户空间下载(或上载)用户定义的字符转换数组来覆盖原有的数组，也可以下载(或上载)用户定义的字库。虚拟控制台的 ioctl( )函数 vt_ioctl( )是个堪称巨型的函数，其代码达 700 行之多，提供了各种各样的功能，/usr/bin 目录下的 setfont、setkeycodes、loadkeys 等工具就是建立在这个系统调用的基础上。由于篇幅的限制，我们在这里就不看这个函数了，有兴趣或需要的读者可自行阅读。对于 translations[ ][ ]中的第一个字符转换数组，即 ASCII 字符(或“Latin-1”)的转换数组，所作的转换实际上就是保持原值不变，只不过是从 8 位变成了 16 位。从代码中可以看出，每个字符转换数组的大小是 256，所以只能支持拼音文字，而不能支持 CJKV（中日朝越）等非拼音文字。

总之，现在变量 tc 中是待显示字符的 16 位 Unicode，而变量 c 中则是该字符原来的 8 位代码。我们继续往下看(drivers/char/console.c)：

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( ) > n_tty_receive_buf( ) > n_tty_receive_char( ) > echo_char( ) > opost( ) > tty_default_put_char( ) > con_write( ) > do_con_write( )]

```
1914
1915                    /* If the original code was a control character we
1916                     * only allow a glyph to be displayed if the code is
1917                     * not normally used (such as for cursor movement) or
1918                     * if the disp_ctrl mode has been explicitly enabled.
1919                     * Certain characters (as given by the CTRL_ALWAYS
1920                     * bitmap) are always displayed as control characters,
1921                     * as the console would be pretty useless without
1922                     * them; to display an arbitrary font position use the
1923                     * direct-to-font zone in UTF-8 mode.
1924                     */
1925            ok = tc && (c >= 32 ||
1926                            (!utf && !(((disp_ctrl ? CTRL_ALWAYS
1927                                       : CTRL_ACTION) >> c) & 1)))
1928                        && (c != 127 || disp_ctrl)
```

```
1929                         && (c != 128+27);
1930
1931         if (vc_state == ESnormal && ok) {
1932             /* Now try to find out how to display it */
1933             tc = conv_uni_to_pc(vc_cons[currcons].d, tc);
1934             if ( tc == -4 ) {
1935                     /* If we got -4 (not found) then see if we have
1936                        defined a replacement character (U+FFFD) */
1937                 tc = conv_uni_to_pc(vc_cons[currcons].d, 0xfffd);
1938
1939                     /* One reason for the -4 can be that we just
1940                        did a clear_unimap( );
1941                        try at least to show something. */
1942                 if (tc == -4)
1943                     tc = c;
1944             } else if ( tc == -3 ) {
1945                     /* Bad hash table -- hope for the best */
1946                 tc = c;
1947             }
1948             if (tc & ~charmask)
1949                                     continue; /* Conversion failed */
1950
1951             if (need_wrap || decim)
1952                 FLUSH
1953             if (need_wrap) {
1954                 cr(currcons);
1955                 lf(currcons);
1956             }
1957             if (decim)
1958                 insert_char(currcons, 1);
1959             scr_writew(himask ?
1960       ((attr << 8) & ~himask)+((tc & 0x100) ? himask : 0)+(tc & 0xff) :
1961                        (attr << 8) + tc,
1962                      (u16 *) pos);
1963             if (DO_UPDATE && draw_x < 0) {
1964                 draw_x = x;
1965                 draw_from = pos;
1966             }
1967             if (x == video_num_columns - 1) {
1968                 need_wrap = decawm;
1969                 draw_to = pos+2;
1970             } else {
1971                 x++;
1972                 draw_to = (pos+=2);
1973             }
1974             continue;
1975         }
1976         FLUSH
```

```
1977                do_con_trol(tty, currcons, c);
1978            }
1979            FLUSH
1980            spin_unlock_irq(&console_lock);
1981
1982    out:
1983            if (from_user) {
1984                /* If the user requested something larger than
1985                 * the CON_BUF_SIZE, and the tty is not stopped,
1986                 * keep going.
1987                 */
1988                if ((orig_count > CON_BUF_SIZE) && !tty->stopped) {
1989                    orig_count -= CON_BUF_SIZE;
1990                    orig_buf += CON_BUF_SIZE;
1991                    count = orig_count;
1992                    buf = orig_buf;
1993                    goto again;
1994                }
1995
1996                up(&con_buf_sem);
1997            }
1998
1999            return n;
2000    #undef FLUSH
2001    }
```

并不是所有的字符都可以显示，所以 1925 行根据字符转换前后的数值确定是否可以显示。如果转换前字符的数值小于 32，即“空格”，那就是控制字符，一般情况下是不能显示的。如果字符可以显示(ok 为 1)，虚拟控制台又处于正常工作状态，就可以进一步处理字符的显示了。

对于虚拟控制台，系统的图形卡工作于字符模式，最终写入图形卡的是一个 16 位短字，其低字节是字符本身的代码，高字节则为字符的“属性”。字符的“属性”决定着显示该字符时的亮度、颜色等要素。所以，最后还得把字符转换成用于图形卡的 8 位代码，这种转换是由 conv_uni_to_pc( )完成的，其代码在 drivers/char/consolemap.c 中：

```
635    int
636    conv_uni_to_pc(struct vc_data *conp, long ucs)
637    {
638        int h;
639        u16 **p1, *p2;
640        struct uni_pagedir *p;
641
642        /* Only 16-bit codes supported at this time */
643        if (ucs > 0xffff)
644            ucs = 0xfffd;        /* U+FFFD: REPLACEMENT CHARACTER */
645        else if (ucs < 0x20 || ucs >= 0xfffe)
646            return -1;        /* Not a printable character */
```

```
647         else if (ucs == 0xfeff || (ucs >= 0x200a && ucs <= 0x200f))
648             return -2;          /* Zero-width space */
649         /*
650          * UNI_DIRECT_BASE indicates the start of the region in the User Zone
651          * which always has a 1:1 mapping to the currently loaded font.  The
652          * UNI_DIRECT_MASK indicates the bit span of the region.
653          */
654         else if ((ucs & ~UNI_DIRECT_MASK) == UNI_DIRECT_BASE)
655             return ucs & UNI_DIRECT_MASK;
656 
657         if (!*conp->vc_uni_pagedir_loc)
658             return -3;
659 
660         p = (struct uni_pagedir *)*conp->vc_uni_pagedir_loc;
661         if ((p1 = p->uni_pgdir[ucs >> 11]) &&
662             (p2 = p1[(ucs >> 6) & 0x1f]) &&
663             (h = p2[ucs & 0x3f]) < MAX_GLYPH)
664             return h;
665 
666         return -4;      /* not found */
667     }
```

这个函数的主体是 660~664 行。虚拟控制台的 vc_data 数据结构中还有个指针 vc_uni_pagedir_loc，指向一个 uni_pagedir 数据结构，这是在 drivers/char/consolemap.c 中定义的：

```
174     struct uni_pagedir {
175         u16           **uni_pgdir[32];
176         unsigned long   refcount;
177         unsigned long   sum;
178         unsigned char   *inverse_translations[4];
179         int       readonly;
180     };
```

对于每个可显示的字符，图形卡在显示器上显示的是这个字符的特殊图形，称为“字模”(glyph)。工作于字符模式(而不是图像模式)的 VGA 图形卡可以显示 256 个字模(所以只能用于拼音文字)。根据正在使用的语言，可以将不同的字模装入图形卡，然后将在这种语言中可以显示的字符的 Unicode 变换成这些字模的编号。这样，就可以达到“本地化”的目的。当然，这只是对采用拼音文字的地区和国家而言。这个数据结构以及 conv_uni_to_pc( )的作用就是实现从 Unicode 到字模编号的变换，或者说映射。对于 ASCII 码，字模的编号与字符的代码相同，但是一般而言都需要加以变换。最简单的变换当然是采用数组的方法，但是对于 16 位的 Unicode 这意味着数组的大小为 64K。这显然是没有必要的，因为对于任何一种具体的拼音文字这个数组都必定是非常稀疏的。为了提高空间效率，人们把 16 位的 Unicode 空间划分成 32 个“码页”，这里指针数组 uni_pgdir[ ]中的每个指针就指向一个码页，以 16 位 Unicode 编码中的最高 5 位作为下标。这样，如果对于具体的语言某个码页为全空，就不必为之分配空间，而只要使相应的指针为 0 即可。为便于管理，每个码页又划分成 32 个子页，再以次 5 位作为下标，

所以 uni_pgdir[ ]中的指针所指向的是二维数组。这些数组的内容是在初始化时通过一个函数 con_set_default_unimap( )设置的，设置的依据是一个临时性的大数组 dfont_unitable[ ]，初始化完成以后，这个数组所占的空间就回收另作它用了。那么，数组 dfont_unitable[ ]的内容又是什么，是从哪里来的呢？显然，其内容应该取决于使用的是哪一种语言。对于英语(美国英语)，drivers/char 目录下有个文件 cp437.uni（表示 437 号码页的 Unicode 编码），我们在这里列出其中一些片断，让读者有个印象：

```
1     #
2     # Unicode table for IBM Codepage 437.  Note that there are many more
3     # substitutions that could be conceived (for example, thick-line
4     # graphs probably should be replaced with double-line ones, accented
5     # Latin characters should replaced with their nonaccented versions,
6     # and some upper case Greek characters could be replaced by Latin), however,
7     # I have limited myself to the Unicodes used by the kernel ISO 8859-1,
8     # DEC VT, and IBM CP 437 tables.
9     #
10    # ---------------------------------
11    #
12    # Basic IBM dingbats, some of which will never have a purpose clear
13    # to mankind
14    #
15    0x00    U+0000
16    0x01    U+263a
17    0x02    U+263b
18    0x03    U+2665
      . . . . . .
47    #
48    # The ASCII range is identity-mapped, but some of the characters also
49    # have to act as substitutes, especially the upper-case characters.
50    #
51    0x20    U+0020
52    0x21    U+0021
53    0x22    U+0022 U+00a8
      . . . . . .
84    0x41    U+0041 U+00c0 U+00c1 U+00c2 U+00c3
85    0x42    U+0042
86    0x43    U+0043 U+00a9
87    0x44    U+0044
      . . . . . .
285   #
286   # Square bullet, non-spacing blank
287   # Mapping U+fffd to the square bullet means it is the substitution
288   # character
289   #
290   0xfe    U+25a0 U+fffd
291   0xff    U+00a0
```

例如，字模 0x42（“B”）的 Unicode 就是 0x42；而 0x41（“A”）则同时对应着 5 个 Unicode 代码。文件中定义的字模编号最高为 0xff，所以只定义了 256 个字模。当编译 Linux 内核时，通过一个工具 conmakehash 从这个文件为 dfont_unitable[ ]生成一个文件 defkeymap.c，后者则是内核源代码的一个组成部分。为帮助读者理解这一点，我们看一下 drivers/char 下面的 Makefile：

```
19      FONTMAPFILE = cp437.uni
        . . . . . .
489     consolemap_deftbl.c: $(FONTMAPFILE) conmakehash
490         ./conmakehash $(FONTMAPFILE) > consolemap_deftbl.c
```

假定现在是在法国，要让内核显示法文，那就要提供用于法文的.uni 文件，并修改 Makefile 中 FONTMAPFILE 的定义，再编译内核就可以了。至于工具 conmakehash，其源代码 conmakehash.c 也在 drivers/char 中。

最终通过 scr_writew( )写入图形卡的是一个 16 位短字，其低字节是字符本身的代码 tc，高字节则为字符的“属性”attr。这里 scr_writew( )是个宏操作，定义于 include/linux/vt_buffer.h：

```
23      #define scr_writew(val, addr) (*(addr) = (val))
```

PC 机图形卡(如 VGA)上的 RAM 区间在 0xa0000 以上、BIOS 以下的位置上，其相对于起点的地址与显示屏上的位置有着一一对应的关系，CPU 可以通过访内指令直接访问。由于图形卡工作于字符方式，所以只要把字符的代码(实际上是字模的编号)连同其属性写入相应的存储单元即可，而不必关心具体的字模和像素。

显然，Linux 内核在“本地化”方面的努力只限于拼音文字，而不适用于 CJKV。以汉字为例，如果要在内核中支持汉字，图形卡就必须工作于图像方式，这样才能显示数千个常用汉字的字模。此外，即使是拼音文字，现代的图形用户界面也要求采用图像方式。不过从字符方式前进到图像方式这一步属于图像处理的范畴，我们这里就从略了。有兴趣或需要的读者可以自己钻研 drivers/video 目录下与“帧缓冲区”(frame buffer)有关的源代码。

回到 n_tty_receive_char( )的代码中，我们已经看了对一般字符的回打，但是还剩下对一些特殊字符的处理，所以继续往下看(drivers/char/n_tty.c)：

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( ) > n_tty_receive_buf( ) > n_tty_receive_char( )]

```
553         if (c == '\r') {
554             if (I_IGNCR(tty))
555                 return;
556             if (I_ICRNL(tty))
557                 c = '\n';
558         } else if (c == '\n' && I_INLCR(tty))
559             c = '\r';
560         if (I_IXON(tty)) {
561             if (c == START_CHAR(tty)) {
562                 start_tty(tty);
563                 return;
```

```
564                 }
565                 if (c == STOP_CHAR(tty)) {
566                     stop_tty(tty);
567                     return;
568                 }
569             }
570             if (L_ISIG(tty)) {
571                 int signal;
572                 signal = SIGINT;
573                 if (c == INTR_CHAR(tty))
574                     goto send_signal;
575                 signal = SIGQUIT;
576                 if (c == QUIT_CHAR(tty))
577                     goto send_signal;
578                 signal = SIGTSTP;
579                 if (c == SUSP_CHAR(tty)) {
580     send_signal:
581                     isig(signal, tty, 0);
582                     return;
583                 }
584             }
```

这里主要是对“\r”、“\n”以及 XON/XOFF 等字符的处理。这就是说，不管这些字符是否属于 process_char_map，对这些字符总是要进行一些处理，只是方式略有不同。还有，就是几个可以导致向终端的控制进程发出信号的字符，如 Ctrl-C、Ctrl-Z 这些字符。这里还要说明，对所有这些字符的处理都是可以通过系统调用 ioctl( )分别加以设置和选择的，所以 ioctl( )对终端设备的驱动起着特别重要的作用。

至此，我们尚未触及对终端设备输入的“规范”(canonical)模式，下面我们就来看“规范”模式对输入的加工或者“烹调”。

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( ) > n_tty_receive_buf( )
 > n_tty_receive_char( )]

```
585             if (tty->icanon) {
586                 if (c == ERASE_CHAR(tty) || c == KILL_CHAR(tty) ||
587                     (c == WERASE_CHAR(tty) && L_IEXTEN(tty))) {
588                     eraser(c, tty);
589                     return;
590                 }
591                 if (c == LNEXT_CHAR(tty) && L_IEXTEN(tty)) {
592                     tty->lnext = 1;
593                     if (L_ECHO(tty)) {
594                         finish_erasing(tty);
595                         if (L_ECHOCTL(tty)) {
596                             put_char('^', tty);
597                             put_char('\b', tty);
```

```
598                 }
599             }
600             return;
601         }
602         if (c == REPRINT_CHAR(tty) && L_ECHO(tty) &&
603             L_IEXTEN(tty)) {
604             unsigned long tail = tty->canon_head;
605 
606             finish_erasing(tty);
607             echo_char(c, tty);
608             opost('\n', tty);
609             while (tail != tty->read_head) {
610                 echo_char(tty->read_buf[tail], tty);
611                 tail = (tail+1) & (N_TTY_BUF_SIZE-1);
612             }
613             return;
614         }
615         if (c == '\n') {
616             if (L_ECHO(tty) || L_ECHONL(tty)) {
617                 if (tty->read_cnt >= N_TTY_BUF_SIZE-1) {
618                     put_char('\a', tty);
619                     return;
620                 }
621                 opost('\n', tty);
622             }
623             goto handle_newline;
624         }
625         if (c == EOF_CHAR(tty)) {
626                 if (tty->canon_head != tty->read_head)
627                     set_bit(TTY_PUSH, &tty->flags);
628             c = __DISABLED_CHAR;
629             goto handle_newline;
630         }
631         if ((c == EOL_CHAR(tty)) ||
632             (c == EOL2_CHAR(tty) && L_IEXTEN(tty))) {
633             /*
634              * XXX are EOL_CHAR and EOL2_CHAR echoed?!?
635              */
636             if (L_ECHO(tty)) {
637                 if (tty->read_cnt >= N_TTY_BUF_SIZE-1) {
638                     put_char('\a', tty);
639                     return;
640                 }
641                 /* Record the column of first canon char. */
642                 if (tty->canon_head == tty->read_head)
643                     tty->canon_column = tty->column;
644                 echo_char(c, tty);
645             }
```

```
646                 /*
647                  * XXX does PARMRK doubling happen for
648                  * EOL_CHAR and EOL2_CHAR?
649                  */
650                 if (I_PARMRK(tty) && c == (unsigned char) '\377')
651                     put_tty_queue(c, tty);
652 
653             handle_newline:
654                 set_bit(tty->read_head, &tty->read_flags);
655                 put_tty_queue(c, tty);
656                 tty->canon_head = tty->read_head;
657                 tty->canon_data++;
658                 kill_fasync(&tty->fasync, SIGIO, POLL_IN);
659                 if (waitqueue_active(&tty->read_wait))
660                     wake_up_interruptible(&tty->read_wait);
661                 return;
662             }
663         }
664 
665         finish_erasing(tty);
666         if (L_ECHO(tty)) {
667             if (tty->read_cnt >= N_TTY_BUF_SIZE-1) {
668                 put_char('\a', tty); /* beep if no space */
669                 return;
670             }
671             if (c == '\n')
672                 opost('\n', tty);
673             else {
674                 /* Record the column of first canon char. */
675                 if (tty->canon_head == tty->read_head)
676                     tty->canon_column = tty->column;
677                 echo_char(c, tty);
678             }
679         }
680 
681         if (I_PARMRK(tty) && c == (unsigned char) '\377')
682             put_tty_queue(c, tty);
683 
684         put_tty_queue(c, tty);
685     }
```

我们不对这些代码详加解释了，读者现在应该已经有了这种能力。这里我们只是要强调，在“规范”模式下，尽管终端设备的 read_buf[ ]缓冲区中已经有了数据，却并不马上就唤醒可能正在睡眠中等待着要从该终端读出的进程，而要到接收到“\n”字符(有可能从“\r”转换而来，见 557 行)以后，或者接收到 EOF、EOL 等字符时才来唤醒(见 660 行)。相比之下，在非“规范”模式，包括原始模式下，则只要缓冲区中的字节数量达到预先设定的 minimum_to_wake，就会唤醒正在睡眠等待的进程，而

minimum_to_wake 一般是 1。具体地，这发生于从 n_tty_receive_char( )返回到 n_tty_receive_buf( )以后，为便于阅读，我们把有关的几行代码再列出于下(drivers\char\n_tty.c)。

[console_softint( ) > run_task_queue( ) > flush_to_ldisc( ) > n_tty_receive_buf( )]

```
761         if (!tty->icanon && (tty->read_cnt >= tty->minimum_to_wake)) {
762             kill_fasync(&tty->fasync, SIGIO, POLL_IN);
763             if (waitqueue_active(&tty->read_wait))
764                 wake_up_interruptible(&tty->read_wait);
765         }
```

这里的 762 行用于异步 I/O，读者可参阅“系统调用 select( )与异步 I/O 一节”。

读者在前面看到，不光控制台有个 tasklet，键盘本身也有个 bh 函数 kbd_bh( )，不过这个 bh 函数处理的只是键盘上的发光二极管，所以对于键盘的读操作没有什么影响。这个 bh 函数的代码在 drivers/char/keyboard.c 中，我们就不看了。

8.9　通用串行外部总线 USB

8.9.1　USB 总线简介

USB(Universal Serial Bus)总线是 20 世纪 90 年代发展起来的一种“通用串行外部总线”，目前在使用中的基本上是 1998 年公布的 USB 1.1 版。最新的版本是 USB 2.0，目前已有芯片投入市场，但是有关的产品（设备）还不多，而支持 2.0 版的设备驱动程序（以及相应版本的操作系统）则一般还在开发或试运行阶段。

首先，USB 是一种“总线”。与传统的外部设备与主机之间的连接方式不同，它允许将不同种类的外部设备混合连接到同一个接口上。可是，它又与计算机内部的总线(如 PCI 总线)不同，CPU 不能通过访内指令或 I/O 指令直接访问连接在 USB 上的设备，而要通过一个“USB 控制器”，间接地与连接在 USB 上的设备打交道，USB 总线存在于计算机的外部，所以说是外部总线。还有，USB 的信号线一共只有两条（外加两条电源线），线上的信号是串行的，所以是“串行外部总线”。至于说“通用”，那是因为 USB 总线的设计从一开始就考虑到了许多不同种类的外部设备，从低速的键盘和鼠标等“人机交互设备(HID)”，到速度较高的通信设备(如 Modem)和存储设备(如 CDROM)，乃至摄像机和显示器等多媒体设备，只要带有 USB 接口就都可以连接到 USB 总线上，并且可以在计算机带电的条件下“现插现用”(plug and play)。

在传统的计算机系统结构中，接口卡可以看作是相应外部设备的一部分，就好像是外部设备派驻在主机内部的联络员一样。所以，每一个接口一般都只能连接到同一种的设备。虽然通过所谓“菊花链”(daisy chain)方式可以把若干同种设备连接到同一接口上，却不能将不同种类的设备混合连接到同一接口上。显而易见，这种结构的可扩充性很差，因为可以插入主机的接口卡数量总是有限的。事实上，应用中很早就有了在同一接口上混合连接不同设备的要求。例如在并行口上既连接打印机又连接

扫描仪、可能还要再连上一台“Zip 驱动器”，就反映了这样的要求。而且，即使主机中有足够多的插槽，也还是有很多问题，例如中断向量的分配和管理、接口卡的成本等等，甚至因太多的连接电缆挤在一起而造成的困难也是个问题。特别值得一提的是：要插入一块接口卡通常得要打开机盖并且关闭电源，而不能在系统加电的情况下随意增减设备。在这样的条件下，所谓“现插现用”实际上是很难真正成为现实的。随着计算机应用的日益普及，这些问题就愈来愈突出了。

另一方面，更为重要的是：电话设备与计算机的结合以及多媒体技术的发展，也对外部设备的连接与驱动提出了新的要求。在电话设备(以及多媒体设备)中普遍采用所谓“等时”(isochronous)传输方式，通常将一个容量较大的信道按时间划分成若干较小的音频或视频信道。例如“T1”传输线就是将一个 1.54Mb 的信道按时间划分成 8000 个 24 字节的“帧”(frame)，而每个帧的 24 个字节又分别用于 24 个 64Kb 的数字化话音信道，使每个话音信道在每 125 微秒的时间内就能传送一个字节。与传统的传输方式相比，等时传输有两大特点。第一，等时传输有比较严格的时间要求，必须为每个信道维持均匀的流量。例如，如果对音频信号的传输忽快忽慢，那么重放出来的声音就会变成有如磁带录音机转速不稳时那样的怪声怪调。相比之下，传统的计算机外部设备（例如从磁盘读入文件时）则并没有这样的要求。第二，等时传输对误码的要求不高，少量的误码（甚至丢失少量数据）对于音频表现为噪音，对视频则表现为画面上的一些“雪花点”，都还是可以容忍的。所以等时传输不需要有 CRC 校验、奇偶校验一类的检错手段。其实，那怕明知传输中有错，甚至丢失，也不能要求对方重发，因为由重发而造成的延迟很可能反而更不能容忍。反之，传统的计算机外部设备对误码的要求就很高，所以通常都带有检错的手段，如果发现出错就要重发。不过，实际上等时传输总是有一定的缓冲，所以只要是在一定的限度内，时间上略有飘移也是允许的。总之，等时传输基本上是周期性的，但是又并非严格意义上的周期信号，并且收发双方也并不需要严格意义上的同步。例如，在上面讲到的“T1”传输线上，接收方只需知道哪一点是一个帧的起点，却不需要知道到底是哪一个帧的起点，因为相差 10 个帧也不过是 1.25 毫秒，通话的双方都感觉不到。一个固定而不太大的延迟是允许的（就像越洋电话那样）。这样的传输方式就称为等时传输，这并不是新技术，但是以前的计算机外部设备中没有这方面的需求。而在 20 世纪 90 年代初，则由于计算机与电话技术相结合的趋向以及多媒体技术的发展，而使之提到了日程上。以电话设备为例，就需要既有支持容量较大的等时方式传输（用于话音），又支持报文传递（用于控制）的外设接口。同时，这种外设接口仍然要适合于传统的外部设备。

USB 正是在这样的背景下发展起来的。90 年代初，人们已经在设备驱动、数据通信、局部网络以及大规模集成电路等方面积累了丰富的经验，USB 的设计从各方面都吸取了营养，而大规模集成电路技术则为 USB 的实现奠定了物质基础。从 90 年代中期开始生产的 PC 机几乎毫无例外地全都带有 USB 接口。

从功能上讲，主机的 USB 接口既可以通过 USB 电缆直接连接到一台支持 USB 的外部设备（我们称之为“USB 设备”），也可以先连接到一个“USB 集中器”(USB Hub)，再由集中器分叉连接到其他集中器或外部设备，从而形成一种星形结构。每根 USB 电缆的长度是 16 呎(约 5 米)，通过集中器级连时最多可以穿越 5 个集中器，从而使最大半径达到 96 呎(约 29 米)。USB 上的信号传输速度有两种，一种是“低速” (Low Speed)，为每秒 1.5 兆位，用于外接键盘、鼠标器等低速设备(常称为 HID，即“人机交互设备”)；另一种是“全速” (Full Speed)，为每秒 12 兆位(作为比较，Ethernet 的速度是每秒 10 兆位)，用于一般的外部设备，包括音频和视频；两种速度的设备可以混用。在后来公布的 USB 标准 2.0 版中，在此基础上又增加了一种传输速度，称为“高速”(High Speed)，达到每秒 480 兆位。USB 设备可以在系统加电的条件下自由地插上或拔下，称为“热插入”。而且，对耗电量很小的设备

或集中器还可以由主机通过 USB 电缆供电。

凡是支持 USB 的外部设备，即 USB 设备(集中器是一种特殊的 USB 设备)，都带有 USB 通信控制器，里面实际上包括了一个微处理器。由此可见，USB 的实现和推广在大规模集成电路技术还不发达、微处理器还比较贵的时代是不现实的。由于主机和 USB 设备都带有控制器，而主机中的控制器起着核心的作用。为了防止混淆，一般把主机一方的控制器称为“主控制器”(Host Controller，常缩写为 HC)。不过，在本书中并不关心设备一方的操作，所以讲到“USB 控制器”时总是指主控制器。除主控制器外，每条 USB 总线一定有个“根集中器”，通常与主控制器集成在同一芯片中。

另一方面，由此也可看出，在某种意义上也可以把 USB 看成是一种计算机网络，一种主/从结构的星形网络。之所以说是主/从结构，是因为信息在 USB 上的传输只能由主机启动，而不能由设备启动，设备永远处于被动的地位。

通过以上的简介，读者已可了解：对于主机系统，USB（实际上是USB控制器）本身也是一项设备，但是对USB操作本身并不是终极的目的，而是为了通过USB对具体的外部设备进行操作。换言之，对USB的操作是手段而不是目的。然而，在另一方面，这种手段却有可能比目的更复杂，因此有必要特别加以介绍。

USB 的信息传输方式是比较特殊的(具有网络和数据通信背景的读者则可能会觉得很自然)。

首先， USB 通过具有一定格式的“信包”(packet)按一定的“规程”(protocol)传输信息，并根据信息(内容)的性质分成如下 4 种传输类型：

(1) 控制型(Control)。主要用于设备的“配置”与控制。信包的容量可以是 8、16、32 或 64 字节，取决于具体的设备(低速设备只支持 8 字节)。在 USB 的整个带宽(传输能力)中，有 10%是为这种信息保留的。这就是说，只要有足够多的控制型信息等着要传递，就必须保证有 10%的带宽用于这些信息，其他信息再多也不能把这部分带宽给挤掉。但是，如果没有足够多的控制型信息要传递，则可以把这部分带宽用于其他类型的信息。控制型信包的传递是带有检错、并须由接收方加以确认的“可靠”传递，如果发现传输出错就要重发。

(2) 等时型(Isochronous)。主要用于实时的音频和视频信号。这种信息是周期性的，又是实时的，对信息传递是否及时有很高的要求，但是对误码却比较能容忍。所以，保证用于等时型信息的带宽是很重要的。USB为等时型信息和中断型信息(也是周期性的)保留90%的带宽(如果有足够多的周期性信息要传递)。另一方面，等时型信息的传递不带检错，也不需要确认，因而就不存在重发的问题。等时型传输的信包通常较大，最高可达1023字节。

(3) 中断型(Interrupt)。名曰“中断”型，实际上却是用于对 USB 设备的周期性查询。USB 设备不存在主动向主机发送“中断请求”的能力，只能被动地受主机通过 USB 总线查询。中断型信息的传递既有时间上的要求，又必须是可靠传递，但是信包较小。信包大小与控制型信包相同。中断型信息和等时型信息二者合在一起不能超过 USB 总线带宽的 90%。

(4) 成块型(Bulk)。用于信息量相对较大，没有很强的时间要求，但是要求可靠传递(带有检错，接收方须确认)的信息。对成块型信息的传递在时间上是没有保证的，USB 总线不为成块型信息保留带宽，只是在执行了前三种传输以后还有时间剩余时才来执行成块型传输。信包的容量取决定于具体的设备，但最大不超过 1023 字节。

同时，USB 控制器把总线上的时间划分成固定大小的“frame”，即“框架”（我们把 frame 这个词翻译成“框架”而不是“帧”，一来与网络技术中的“帧”有所区别，二来意义上也似乎更为贴切）。

每个框架的大小是 1 毫秒。在每个框架中主机都可以与 USB 总线上的设备相互发送许多信包，不过在每一特定的时刻只能在一个方向上发送，所以是“半双工”链路。主机与设备之间的通信只能由主机启动，而设备则永远处于从属的、被动的地位。因而主机与设备间是主/从关系。在每一个框架内可以有许多个主机与设备间的“交互”(Transaction)，即二者间若干信包的交换。每个信包传递的方向可以是从主机到设备，也可以是从设备到主机，但是每个交互中的第一个信包总是从主机到设备，因为只有主机可以发起一次交互。信包的大小因传输方式和设备类型而异，对于全速设备的控制(型)传输最大为 64 字节，对低速设备则不超过 8 个字节，但是等时(型)传输的信包可达 1023 字节。每个框架的开头总是由主机向总线上的所有设备广播一个特殊的“框架开始”(SOF)信包，作为一种同步手段。框架的划分是硬性的、没有弹性的，时候一到，主机的 USB 控制器就发出 SOF 信包。

主机与USB总线上各设备之间的位同步，靠调制在信号波形中的时钟脉冲实现，框架的同步靠SOF信包实现，而信包的同步则靠信包头部的特殊格式实现。

一般而言，主机与设备之间的信息传递又可以根据其目的分成两种。一种是应用信息的传递，包括数据以及由具体设备规定的一些应用层上的控制/状态信息，例如待打印的字符串，又如对扫描器的“开启光源”命令、“开始扫描”命令等等，就属于这一种。从传输的角度看，这些信息对于 USB 总线是“透明”的，USB 总线只是把这些信息以信包的形式作为无格式字节流传递给对方，而并不关心其内容，也不规定其格式。当然，应用软件以及具体设备的驱动程序可以自己规定这些信息的格式。

另一种是 USB 总线为维持其本身的正常运行和管理所需的一些“USB 层”的控制/状态信息，例如，为要了解某一设备是否还在总线上运行，或者要了解某一设备都具有哪些功能，就需要从设备读入其作为 USB 设备必须提供的状态信息，等等。我们有时称此种控制/状态信息为“高层”控制/状态信息，之所以称为“高层”是为了与下面所说的低层控制/状态信息相区别。USB 总线为这些信息规定了特定的格式，例如后面要讲到的设备描述体、接口描述体等等。不过，根据具体的情况，在传递控制/状态信息时往往也可以附加传递一些应用信息。

本来，USB 总线上信息传输的目的就在于传递这些应用信息和“高层”控制/状态信息。可是，为了启动传递这些信息以及保证这些信息得到可靠的传递，还需要传递一些附加的信息。这些信息的传递本身并不是目的，而只是为达到上述目的而采取的手段。

每次高层控制/状态信息或者(以及)应用信息的传递称为一次“传输”(transfer)，因信息类型的不同可以分成上述控制、中断、等时以及成块 4 种。同时，按传输的方向又可分成输入和输出两种。每次传输都由一次或数次“交互”构成。每次交互又包括三个信包(等时传输的交互例外、只有两个信包)的传递，其中第一个信包总是由主机发出的低层控制信息，称为“传令”(token)信包，信包的内容表明了交互的对象以及后续信包的传递方向；然后是一个载运着应用信息或高层控制/状态信息的信包，称为“数据”(data)信包；最后则是由数据信包的接收方发出载运着确认信息的信包，称为“握手”(handshake)信包。不过，在等时传输方式的交互中接收方不发出握手信包，所以只有两个信包。可见，传令信包和握手信包所载运的信息是为完成交互所需的“低层”控制/状态信息，在性质上有别于上述的高层控制/状态信息。高层控制/状态信息的目的在于维持 USB 总线的正常运行和管理，而低层控制/状态信息的目的在于保证应用信息或高层控制/状态信息的正确传递。每个交互只传递一个数据信包，数据信包传递的方向即为交互的方向，往往也就是传输的方向。交互是一个不可分割的整体，其三个(或两个)信包的传递必须在同一框架中完成。在主机发出传令信包，启动了一次交互以后，目标设备必须立即(在规定的时间内)作出反应，或从主机接收数据信包，或将数据信包发送给主机，否则本次交互便失败了。而一次传输，如果包含多次交互，则可以跨越多个时间框架。在前述的 4 种传输方式中，成

块、等时和中断三种应用信息的传输都只包含一种(虽然可能多个)交互，但是控制(型)传输则一般需要通过三种不同性质的交互才能完成，称为控制传输的三个“阶段”。其中第一个阶段为“SETUP”阶段，只有一个交互，在这个交互中传递的是高层的控制命令，如“读配置信息”、“设置地址”等等。第二个阶段是数据阶段，根据需要可以有多个交互，也可以没有。在这些交互中传递的是与本次命令相联系的信息，也可以是一些应用信息；如果不需要传递这样的信息，则也可以跳过数据阶段。第三个阶段是状态阶段，这个交互中传递的是与操作目标有关的状态信息。相比之下，成块、等时和中断三种传输都只有一个阶段，即数据阶段。

总之，USB 总线上的信息流通都以信包的形式进行。按照信包的性质和作用，可以分成下列几类：

(1) 传令(Token)。由主机发出，用来启动一次交互。除本次交互目标的地址外，信包中还含有交互的方向和性质，IN 表示输入，OUT 表示输出，SETUP 则用于控制传输的启动。
(2) 数据。根据交互的方向，由主机或设备发出，其内容为应用信息或高层控制/状态信息。
(3) 握手。由数据信包或传令信包的接收方(或集中器)发出，说明对数据信包或传令信包的接收情况，ACK 表示成功，NAK 表示出错或无反应，STALL(由集中器发出)表示不能接收。
(4) SOF。用于框架的分隔，信包中含有框架的序号。
(5) 低速设备前缀。低速设备在每个信包之前都要加上一个特殊的前缀。

每个物理的 USB 设备上可以有一个或多个“功能”，相当于“逻辑设备”。在有些设备中，这些功能的划分和组合是可以改变的，此时一种特定的划分和组合就称为一种“配置”(configuration)。在这方面，电话通信设备就是很典型的。以前述的 T1 设备为例(中国采用 E1，有 32 个 8KB 数字化信道)，其 24 个数字化信道就可以按需要加以组合，例如将其中的 12 个信道用于电话，而拿出 8 个信道组合成一个 64KB 的信道用于互联网，剩下 4 个信道则用作通向 4 个营业所的专线连接，这就是一个具体的配置。同时，为了与主机通信的需要，在 USB 设备中设置了一些“端点”(endpoint)。每个端点只支持一种性质的传输。除用于控制传输的端点为整个设备所公用外，其他端点都属于某个具体的功能，即逻辑设备。反过来，根据具体的需要，每个功能可以有多个通信端点。在某些设备中，这些通信端点的从属就像功能的划分一样是可以改变的。

主机与具体 USB 设备的连系在概念上按传输类型和对象的不同而分成许多“管道”(Pipe)。如前所述，每个 USB 设备可以有若干个“功能”，每个功能可以提供若干通信“端口”(Port)，而主机与每个端口之间就是一个逻辑上的管道（注意不要与进程间通信管道混淆）。

主机负有调度USB总线上信息传输的责任。首先是把系统中的等时传输流量分配给各个时间框架。所以，每个等时传输中的各个交互明确地属于某个时间框架。每条USB总线有1024个时间框架，相应地就有1024个等时交互队列。队列中的每个数据结构（称为“交互描述块”）都代表着一个属于等时传输的交互请求。USB总线控制器每一秒钟扫描一遍所有这些队列。所以，每个队列在每一秒钟的时间内得到一次执行。其余的三种传输请求都不分配到具体的时间框架，而是根据传输的类型排在各自的队列中。USB总线控制器在执行完每个框架中的等时交互以后就会来执行这些队列中的交互(请求)。不过中断传输（只含一个交互）是以交互为单位挂入队列，而控制传输和成块传输则以传输为单位挂入队列，每个控制传输或成块传输本身又是一个交互请求的队列（即使队列中只有一个交互）。

完成了对各种交互请求的调度，建立起相应的数据结构和队列以后，CPU 就完成了任务，以后就是 USB 总线控制器的事了。在运行中，USB 总线控制器根据其内部对时钟脉冲的计数确定在什么时候

开始一个框架以及哪一个框架，接着就“执行”该框架的等时交互队列。然后，执行完一个框架的等时交互以后，还要执行若干中断交互。到执行完一个框架的等时交互和中断交互时，应该至少还有10%的时间剩余，此时就来执行其他队列，一直到框架中剩余的时间已不足以完成一次交互、而只好停下来等待下一个框架为止。对于控制传输和成块传输的交互请求队列（也就是对每个控制传输或成块传输），执行时有两种不同的方式。一种称为“纵向执行”，就是先顺着一个队列执行完所有的交互请求，再开始另一个队列；另一种称为“横向执行”，就是在一个队列中执行一个（或几个）交互请求，然后就转到下一个队列中。等时交互和中断交互在执行以后仍留在队列中，而控制交互和成块交互则在执行以后由 USB 总线控制器自动予以脱链。CPU 仅在需要改变调度或处理传输结果时才需要介入。

可见，USB 上的信息传输，尤其是主机一侧的过程，是个相当复杂、并且时间性要求很高的过程，需要硬件和软件分工合作才能完成。那么怎样分工呢？这就需要在硬件和软件之间划出一个标准的“主机控制器界面”。目前有两种这样的界面。一种是由 Intel 制定的，称为“Universal Host Controller Interface”，缩写为 UHCI；另一种是由 Compaq、Microsoft 以及 National Semiconductor 联合制定的，称为“Open Host Controller Interface”，缩写为 OHCI。大体上说，OHCI 把更多的功能划到硬件一边，从而相应的芯片就更复杂一些，软件则可以简单一些；而 UHCI 则让软件方面多承担一些，对硬件的要求就相对低一些。不言而喻，控制器芯片的不同当然会导致驱动软件的不同，但是这种不同只存在于驱动软件的低层。所以，USB 总线的驱动程序又分成两层，其中“USB 层”是共同的；在它下面则是“HC”层，根据所用的芯片而选用 UHCI 或 OHCI 驱动程序。Linux 内核对二者都提供支持，可以通过条件编译选择项选用。对 OHCI 的实现在 drivers/usb/usb-ohci.c 和 drivers/usb/usb-ohci.h 两个文件中。对 UHCI 的实现则又有两种，也是通过条件编译选择项选用。其中之一在 drivers/usb/usb-uhci.c 和 drivers/usb/usb-uhci.h 两个文件中，另一种则在 drivers/usb/uhci.c 和 drivers/usb/uhci.h 两个文件中。我们在本节中阅读代码时采用后者，但是读者应该不难举一反三。我们之所以选用 UHCI，是因为 UHCI 的软件更复杂一些，但是读了以后对 USB 的理解会更深一些。目前，Linux 内核 2.4.0 版所实现的是 USB 1.1 版，以后肯定很快就会支持 USB 2.0 版。

除 USB 以外，大约在同一时期还发展起了另一种类似的串行外设总线，称为“Firewire”。后来由 IEEE 加以标准化，成了 IEEE1394 标准，Linux 内核也支持 IEEE1394。但是，从市场的情况来看，USB 已经得到更为广泛的应用，并已进入良性循环，所以也更为重要，而且二者在逻辑上颇为相似。本书既然花了较大的篇幅介绍 USB，就不再涉及 IEEE1394 了。内核中与此有关的代码基本上都在 drivers/ieee1394 下面，有兴趣或需要的读者可以自己加以研究。

此外，还要提一下并行外设总线。原先的“并行口”也已发展成为并行外设总线，并且也已由 IEEE 加以标准化，成为了 IEEE1284 标准；同样，Linux 内核也支持 IEEE1284。相比之下，IEEE1284 比 USB 简单，功能也没有 USB 强，使用也没有 USB 方便，估计可能会慢慢被 USB 取代。限于本书的篇幅，我们也不能在此介绍 IEEE1284 了。与此有关的几个文件在 drivers/parport 下面，读者不妨自己阅读。

从串行口和并行口朝着串行外设总线和并行外设总线的发展过程中，我们可以看出在传统的设备驱动层中正在形成一个新的子层。使用这些总线以后，原来的“底层”驱动程序现在要依靠相应总线驱动程序所提供的服务才能访问目标设备了。似乎可以说：内核在设备驱动方面的重点正在从提供直接的设备驱动向提供实现设备驱动的手段和环境转移。

8.9.2 USB 总线的初始化和 USB 设备的枚举

我们先看 USB 总线本身的初始化。首先，USB 控制器(连同根集中器)连接在 PCI 总线上，是一个 PCI 设备，在 PCI 总线的初始化过程中也会受到枚举。PCI 设备的初始化完成以后，在 PCI 总线树中就有了代表着具体 USB 总线控制器的 pci_dev 数据结构，并已为控制器的 I/O 区间和 RAM 区间分配和设置了总线地址。

在 USB 总线控制器的设备驱动程序方面，则要为其准备下一个 pci_driver 数据结构，其类型定义在 include/linux/pci.h 中：

```
449     struct pci_driver {
450         struct list_head node;
451         char *name;
452         const struct pci_device_id *id_table;   /* NULL if wants all devices */
453         int  (*probe)(struct pci_dev *dev,  const struct pci_device_id *id);
                                                    /* New device inserted */
454         void  (*remove)(struct pci_dev *dev);    /* Device removed (NULL if not
                                                       a hot-plug capable driver) */
455         void  (*suspend)(struct pci_dev *dev);   /* Device suspended */
456         void  (*resume)(struct pci_dev *dev);    /* Device woken up */
457     };
```

这个数据结构为通用的 PCI 设备管理机制提供了几个函数指针，特别是为一个通用的、一般化的 PCI 设备初始化过程提供了函数指针 probe。供这个 PCI 设备的初始化过程“回叫”，以完成具体设备的初始化。对于遵循 UHCI 界面的 USB 控制器，其 pci_driver 数据结构为 uhci_pci_driver，定义于 drivers/usb/uhci.c：

```
2458    static struct pci_driver uhci_pci_driver = {
2459        name:       "usb-uhci",
2460        id_table:   &uhci_pci_ids [0],
2461
2462        probe:      uhci_pci_probe,
2463        remove:     uhci_pci_remove,
2464
2465    #ifdef  CONFIG_PM
2466        suspend:    uhci_pci_suspend,
2467        resume:     uhci_pci_resume,
2468    #endif  /* PM */
2469    };
```

结构中的指针 id_table 应该指向一个 pci_device_id 结构数组，表明由这个数据结构所确定的设备驱动程序适用于哪一些 PCI 设备。对此，drivers/usb/uhci.c 中相应地定义了数组 uhci_pci_ids[ ]:

```
2441    static const struct pci_device_id __devinitdata uhci_pci_ids [ ] = { {
2442
2443        /* handle any USB UHCI controller */
2444        class:      ((PCI_CLASS_SERIAL_USB << 8) | 0x00),
2445        class_mask:     ~0,
2446
2447        /* no matter who makes it */
2448        vendor:     PCI_ANY_ID,
2449        device:     PCI_ANY_ID,
2450        subvendor:  PCI_ANY_ID,
2451        subdevice:  PCI_ANY_ID,
2452
2453        }, { /* end: all zeroes */ }
2454    };
```

从其定义可以看出，它适用于所有类型为 PCI_CLASS_SERIAL_USB，子类型为 0 的 PCI 设备，即 USB 控制器，而不管是由哪一家厂商提供。

准备好这些数据结构以后，就可以通过 inline 函数 pci_module_init( )向系统登记具体的设备驱动程序，并对设备进行初始化。其代码在 include/linux/pci.h 中：

```
602    /*
603     * a helper function which helps ensure correct pci_driver
604     * setup and cleanup for commonly-encountered hotplug/modular cases
605     *
606     * This MUST stay in a header, as it checks for -DMODULE
607     */
608    static inline int pci_module_init(struct pci_driver *drv)
609    {
610        int rc = pci_register_driver (drv);
611
612        if (rc > 0)
613            return 0;
614
615        /* iff CONFIG_HOTPLUG and built into kernel, we should
616         * leave the driver around for future hotplug events.
617         * For the module case, a hotplug daemon of some sort
618         * should load a module in response to an insert event. */
619    #if defined(CONFIG_HOTPLUG) && !defined(MODULE)
620        if (rc == 0)
621            return 0;
622    #endif
623
624        /* if we get here, we need to clean up pci driver instance
625         * and return some sort of error */
626        pci_unregister_driver (drv);
627
628        return -ENODEV;
```

```
629     }
```

这里通过 pci_register_driver( )完成 PCI 设备驱动程序的登记和初始化，这就是前面所讲的通用、一般化的 PCI 设备初始化过程。这个函数返回一个计数，表示在 PCI 总线上找到了几个这样的设备。一般，如果这个函数返回 0 就要调用 pci_unregister_driver( ) 撤销登记，但是如果 PCI 总线允许“热插入”。即在加电以后的运行过程中带电插入设备，而驱动程序又并非通过可安装模块实现，则不应撤销登记(621 行)；因为以后热插入此种设备时仍需要执行这个驱动程序，应该保留着，以备热插入此种设备之需。函数 pci_register_driver( )的代码在 drivers/pci/pci.c 中：

[pci_module_init( ) > pci_register_driver( )]

```
324     int
325     pci_register_driver(struct pci_driver *drv)
326     {
327         struct pci_dev *dev;
328         int count = 0;
329 
330         list_add_tail(&drv->node, &pci_drivers);
331         pci_for_each_dev(dev) {
332             if (!pci_dev_driver(dev))
333                 count += pci_announce_device(drv, dev);
334         }
335         return count;
336     }
```

首先将 USB 总线控制器的 pci_driver 数据结构链入内核中的队列 pci_drivers，这就是所谓“登记”。然后通过一个循环对所有的 pci_dev 数据结构调用 pci_dev_driver( )，统计一下有多个 PCI 设备尚未与具体的驱动程序挂上钩。这个函数的代码在 drivers/pci/pci.c 中：

[pci_module_init( ) > pci_register_driver( ) > pci_dev_driver( )]

```
456     struct pci_driver *
457     pci_dev_driver(const struct pci_dev *dev)
458     {
459         if (dev->driver)
460             return dev->driver;
461         else {
462             int i;
463             for(i=0; i<=PCI_ROM_RESOURCE; i++)
464                 if (dev->resource[i].flags & IORESOURCE_BUSY)
465                     return &pci_compat_driver;
466         }
467         return NULL;
468     }
```

如果一个设备的 pci_dev 结构尚未与任何驱动程序挂钩，并且其所有地址区间都尚未启用，则 pci_dev_driver( )返回 0。这样的 pci_dev 结构需要通过 pci_announce_device( )加以比对(drivers/pci/pci.c)。

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( )]

```
299	static int
300	pci_announce_device(struct pci_driver *drv, struct pci_dev *dev)
301	{
302		const struct pci_device_id *id;
303		int ret = 0;
304	
305		if (drv->id_table) {
306			id = pci_match_device(drv->id_table, dev);
307			if (!id) {
308				ret = 0;
309				goto out;
310			}
311		} else
312			id = NULL;
313	
314		dev_probe_lock( );
315		if (drv->probe(dev, id) >= 0) {
316			dev->driver = drv;
317			ret = 1;
318		}
319		dev_probe_unlock( );
320	out:
321		return ret;
322	}
```

可想而知，所谓比对是将具体设备的类型、厂家等等在 PCI 枚举阶段从设备收集的信息与 USB 驱动程序的数组 uhci_pci_ids[ ]进行比对，具体是由 pci_match_device( )完成的(drivers/pci/pci.c):

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > pci_match_device( )]

```
284	const struct pci_device_id *
285	pci_match_device(const struct pci_device_id *ids, const struct pci_dev *dev)
286	{
287		while (ids->vendor || ids->subvendor || ids->class_mask) {
288			if ((ids->vendor == PCI_ANY_ID || ids->vendor == dev->vendor) &&
289			    (ids->device == PCI_ANY_ID || ids->device == dev->device) &&
290			    (ids->subvendor == PCI_ANY_ID ||
				ids->subvendor == dev->subsystem_vendor) &&
291			    (ids->subdevice == PCI_ANY_ID ||
				ids->subdevice == dev->subsystem_device) &&
292			    !((ids->class ^ dev->class) & ids->class_mask))
293				return ids;
```

```
294             ids++;
295         }
296         return NULL;
297     }
```

如果比对结果相符，就找到了一个 USB 总线控制器，此时便通过驱动程序提供的函数指针 probe 对其进行初始化。对于遵循 UHCI 界面的 USB 总线控制器，这个函数是 uhci_pci_probe( )，其代码在 drivers/usb/uhci.c 中：

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( )]

```
2376    static int __devinit uhci_pci_probe(struct pci_dev *dev,
                                             const struct pci_device_id *id)
2377    {
2378        int i;
2379
2380        /* disable legacy emulation */
2381        pci_write_config_word(dev, USBLEGSUP, 0);
2382
2383        if (pci_enable_device(dev) < 0)
2384            return -ENODEV;
2385
2386        if (!dev->irq) {
2387            err("found UHCI device with no IRQ assigned. check BIOS settings!");
2388            return -ENODEV;
2389        }
2390
2391        /* Search for the IO base address.. */
2392        for (i = 0; i < 6; i++) {
2393            unsigned int io_addr = pci_resource_start(dev, i);
2394            unsigned int io_size = pci_resource_len(dev, i);
2395
2396            /* IO address? */
2397            if (!(pci_resource_flags(dev, i) & IORESOURCE_IO))
2398                continue;
2399
2400            /* Is it already in use? */
2401            if (check_region(io_addr, io_size))
2402                break;
2403
2404            pci_set_master(dev);
2405            return setup_uhci(dev, dev->irq, io_addr, io_size);
2406        }
```

首先通过 pci_enable_device( )及其下面的一系列子程序启用作为 PCI 设备的 USB 控制器。这些函数都在 drivers/pci/pci.c 或 arch/i386/kernel/pci-i386.c 中，我们把它们留给读者自己阅读，作为对 PCI 总

线一节的复习。

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > pci_enable_device( )]

```
242	/**
243	 * pci_enable_device   Initialize device before it's used by a driver.
244	 * @dev: PCI device to be initialized
245	 *
246	 *  Initialize device before it's used by a driver. Ask low-level code
247	 *  to enable I/O and memory. Wake up the device if it was suspended.
248	 *  Beware, this function can fail.
249	 */
250	int
251	pci_enable_device(struct pci_dev *dev)
252	{
253		int err;
254	
255		if ((err = pcibios_enable_device(dev)) < 0)
256			return err;
257		pci_set_power_state(dev, 0);
258		return 0;
259	}
```

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > pci_enable_device( ) > pcibios_enable_device( )]

```
1042	int pcibios_enable_device(struct pci_dev *dev)
1043	{
1044		int err;
1045	
1046		if ((err = pcibios_enable_resources(dev)) < 0)
1047			return err;
1048		pcibios_enable_irq(dev);
1049		return 0;
```

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > pci_enable_device( ) > pcibios_enable_device( ) > pcibios_enable_resources( )]

```
306	int pcibios_enable_resources(struct pci_dev *dev)
307	{
308		u16 cmd, old_cmd;
309		int idx;
310		struct resource *r;
311	
312		pci_read_config_word(dev, PCI_COMMAND, &cmd);
313		old_cmd = cmd;
```

```
314         for(idx=0; idx<6; idx++) {
315             r = &dev->resource[idx];
316             if (!r->start && r->end) {
317                 printk(KERN_ERR
        "PCI: Device %s not available because of resource collisions\n", dev->slot_name);
318                 return -EINVAL;
319             }
320             if (r->flags & IORESOURCE_IO)
321                 cmd |= PCI_COMMAND_IO;
322             if (r->flags & IORESOURCE_MEM)
323                 cmd |= PCI_COMMAND_MEMORY;
324         }
325         if (dev->resource[PCI_ROM_RESOURCE].start)
326             cmd |= PCI_COMMAND_MEMORY;
327         if (cmd != old_cmd) {
328             printk("PCI: Enabling device %s (%04x -> %04x)\n",
                                                    dev->slot_name, old_cmd, cmd);
329             pci_write_config_word(dev, PCI_COMMAND, cmd);
330         }
331         return 0;
332     }
```

我们在前面曾说 USB 设备没有向主机发出中断请求的能力，而只能等待，受主机的查询。但是这并不意味着 USB 控制器(在主机中) 没有向 CPU 发出中断请求的能力，这是完全不同的两回事，不能混淆。事实上，USB 控制器是有能力向 CPU 发出中断请求的，所以要接通它的中断请求线(arch/i386/kernel/pci-irq.c)。

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > pci_enable_device( ) > pcibios_enable_device( ) > pcibios_enable_irq( )]

```
613     void pcibios_enable_irq(struct pci_dev *dev)
614     {
615         u8 pin;
616         pci_read_config_byte(dev, PCI_INTERRUPT_PIN, &pin);
617         if (pin && !pcibios_lookup_irq(dev, 1) && !dev->irq) {
618             char *msg;
619             if (io_apic_assign_pci_irqs)
620                 msg = " Probably buggy MP table.";
621             else if (pci_probe & PCI_BIOS_IRQ_SCAN)
622                 msg = "";
623             else
624                 msg = " Please try using pci=biosirq.";
625             printk(KERN_WARNING
                    "PCI: No IRQ known for interrupt pin %c of device %s.%s\n",
626                    'A' + pin - 1, dev->slot_name, msg);
627         }
628     }
```

USB 控制器本身带有微处理器，在 USB 总线上发送/接收的信息都由 USB 控制器通过 DMA 直接从内存读/写，主机的 CPU 只要提供缓冲区指针就可以了。而且，CPU 也不需要逐次地为 USB 总线上的操作提供缓冲区指针，而只要把缓冲区指针纪录在相应的交互请求，或曰交互描述块中就可以了。USB 控制器自会顺着交互请求队列逐个地完成对这些缓冲区的操作。类似的 DMA 操作称为“智能化 DMA”。其实，USB 总线控制器的 DMA 操作甚至比一般的智能化 DMA 还要复杂，因为 CPU 为之准备的并不只是一个缓冲区队列，而是许多交互请求队列，称为一个“调度”。

PCI 设备(的接口)要进行 DMA 操作就得具有竞争成为“总线主”的能力。另一方面，PCI 设备的 DMA 功能还要服从 CPU 的统一管理，在 PCI 配置寄存器组的命令寄存器中有一个控制位 PCI_COMMAND_MASTER，就是用来打开或关闭具体 PCI 设备竞争成为总线主的能力。在完成 PCI 总线的初始化时，所有 PCI 设备的 DMA 功能都是关闭的，所以这里要通过 pci_set_master( )启用 USB 控制器竞争成为总线主的能力(drivers/pci/pci.c)。

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( )
 > pci_set_master( )]

```
508    void
509    pci_set_master(struct pci_dev *dev)
510    {
511        u16 cmd;
512
513        pci_read_config_word(dev, PCI_COMMAND, &cmd);
514        if (! (cmd & PCI_COMMAND_MASTER)) {
515            DBG("PCI: Enabling bus mastering for device %s\n", dev->slot_name);
516            cmd |= PCI_COMMAND_MASTER;
517            pci_write_config_word(dev, PCI_COMMAND, cmd);
518        }
519        pcibios_set_master(dev);
520    }
```

再回到 uhci_pci_probe( )的代码中，USB 控制器作为 PCI 设备在 PCI 总线层次上的初始化已经完成了，下面就是 USB 总线控制器本身的初始化，即 USB 总线的初始化了。这是由 setup_uhci( )完成的，其代码在 drivers/usb/uhci.c 中：

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( )]

```
2327    /*
2328     * If we've successfully found a UHCI, now is the time to increment the
2329     * module usage count, and return success..
2330     */
2331    static int setup_uhci(struct pci_dev *dev, int irq,
                                unsigned int io_addr, unsigned int io_size)
2332    {
2333        int retval;
2334        struct uhci *uhci;
```

```
2335        char buf[8], *bufp = buf;
2336
2337    #ifndef __sparc__
2338        sprintf(buf, "%d", irq);
2339    #else
2340        bufp = __irq_itoa(irq);
2341    #endif
2342        printk(KERN_INFO __FILE__ ": USB UHCI at I/O 0x%x, IRQ %s\n",
2343            io_addr, bufp);
2344
2345        uhci = alloc_uhci(io_addr, io_size);
2346        if (!uhci)
2347            return -ENOMEM;
2348        dev->driver_data = uhci;
2349
2350        request_region(uhci->io_addr, io_size, "usb-uhci");
2351
2352        reset_hc(uhci);
2353
2354        usb_register_bus(uhci->bus);
2355        start_hc(uhci);
2356
2357        retval = -EBUSY;
2358        if (request_irq(irq, uhci_interrupt, SA_SHIRQ, "usb-uhci", uhci) == 0) {
2359            uhci->irq = irq;
2360
2361            pci_write_config_word(dev, USBLEGSUP, USBLEGSUP_DEFAULT);
2362
2363            if (!uhci_start_root_hub(uhci))
2364                return 0;
2365        }
2366
2367        /* Couldn't allocate IRQ if we got here */
2368
2369        reset_hc(uhci);
2370        release_region(uhci->io_addr, uhci->io_size);
2371        release_uhci(uhci);
2372
2373        return retval;
2374    }
```

参数 dev 指向 USB 总线控制器的 pci_dev 数据结构，irq 为该 PCI 设备所连接的中断请求输入线号，io_addr 则为其 I/O 地址空间的起始地址，区间的大小为 io_size。

每个 USB 控制器控制着一条 USB 总线，需要有一个数据结构作为代表。对于遵循 UHCI 界面的控制器，那就是 uhci 结构，定义于 drivers/usb/uhci.h：

```
308    struct uhci {
```

```
309         /* Grabbed from PCI */
310         int irq;
311         unsigned int io_addr;
312         unsigned int io_size;
313 
314         struct list_head uhci_list;
315 
316         struct usb_bus *bus;
317 
318         struct uhci_td skeltd[UHCI_NUM_SKELTD]; /* Skeleton TD's */
319         struct uhci_qh skelqh[UHCI_NUM_SKELQH]; /* Skeleton QH's */
320 
321         spinlock_t framelist_lock;
322         struct uhci_framelist *fl;                  /* Frame list */
323         int fsbr;                                   /* Full speed bandwidth reclamation */
324 
325         spinlock_t qh_remove_lock;
326         struct list_head qh_remove_list;
327 
328         spinlock_t urb_remove_lock;
329         struct list_head urb_remove_list;
330 
331         struct s_nested_lock urblist_lock;
332         struct list_head urb_list;
333 
334         struct virt_root_hub rh;    /* private data of the virtual root hub */
335     };
```

数据结构中有几个特别重要的成分。首先是指针 fl，指向一个 uhci_framelist 数据结构，这就是具体 USB 总线的“框架表”，这种数据结构也定义于 drivers/usb/uhci.h：

```
100     struct uhci_framelist {
101         __u32 frame[UHCI_NUMFRAMES];
102     } __attribute__((aligned(4096)));
```

USB 总线的框架表实际上是个指针数组，每个指针都指向一个等时交互队列。常数 UHCI_NUMFRAMES 在同一文件中定义为 1024，所以整个数组代表着 1024 个框架。数组的起始地址必须与 4K 字节边界对齐，这样其起始地址的低 12 位就全都是 0。USB 控制器内部有个“框架表基地址寄存器”，用来记录这个基地址。同时，USB 控制器内部还有个 10 位的“框架计数器”，这个计数器从 0 开始，每过 1 毫秒(1/1024 秒)就加 1，直至 0x3ff 即 1023，然后又变为 0，如此周而复始。在框架计数的后面添上两位 0，再与框架表的基地址连在一起，就成了指向框架表中某个表项的指针。框架表中的每个表项都指向一个 uhci_td 结构的队列，每个 uhci_td 结构是对一个交互的描述，我们称之为“交互描述块”或“交互请求”。USB 控制器在每个框架中首先就执行这个队列。每个等时交互队列中的最后一个数据结构指向一个(实际上是一截)中断交互队列。中断交互队列与框架之间并不是一一对应的关系，uhci 结构中有个 uhci_td 结构数组 skeltd[ ]，其中的每个元素都指向一截中断交互请求队列，常数

UHCI_NUM_SKELTD 定义为 10，其中 skeltd[0]是整个队列的终点，而 skeltd[9]实际上不用，所以一共有 8 截这样的中断交互请求队列。这些中断交互请求队列又链接在一起，成为一个总的中断交互请求队列。但是，链接在不同部位上的中断交互请求受到执行的频率是不一样的。当将一个代表着中断交互的 uhci_td 结构链入队列时，可以根据所要求的执行周期选择链入中断交互请求队列的不同部位。而 skeltd[ ]中的各个元素，则起着链入点的作用，所以这个数组称为中断交互请求队列的“骨架”(skeleton)。这些链入点本身也是 uhci_td 结构，不过是空闲的 uhci_td 结构，USB 控制器在执行时会自动跳过。此外，下面读者将会看到，虽然中断交互请求队列并不与框架一一对应，二者间还是有着某种静态的对应关系。

等时交互和中断交互都是周期性的，在每个框架中二者的流量加在一起不超过 90%。这样，在每个框架中，USB 控制器在执行完这两种交互请求以后总是还有一些时间(至少 10%)，可以用来执行控制交互以及成块交互。这两种交互都不是周期性的，其队列与框架没有静态的对应关系，USB 控制器对这些队列的执行完全是动态的，有时间就执行，没有时间就不执行，时间多就多执行，时间少就少执行。在 uhci 结构中还有个 uhci_qh 结构数组 skelqh[ ]，数组中的每个元素都是一个队列头，用来维持一个“队列的队列”，或者说传输请求的队列。如前所述，每个传输请求是一个交互请求的队列。所以，这个数组是控制/成块传输请求队列的骨架，其大小是 UHCI_NUM_SKELQH，在 drivers/usb/uhci.h 中定义为 4。从逻辑上说，只要有两个链入点就够了，可是实际上 USB 设备有全速和低速之分，还有一个有着特殊的用途，所以共有 4 个。

因此，除还有其他一些成分以外，uhci 数据结构实际上代表着主机 CPU 为一条 USB 总线排好的“日程表”，或者说执行程序，这就称为一个“调度”(schedule)。不言而喻，初始化时要为 USB 控制器分配、建立一个 uhci 数据结构。这是由 alloc_uhci( )完成的，其代码在 drivers/usb/uhci.c 中。这个函数的代码较长，我们分段阅读。

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > alloc_uhci( )]

```
2129    /*
2130     * Allocate a frame list, and then setup the skeleton
2131     *
2132     * The hardware doesn't really know any difference
2133     * in the queues, but the order does matter for the
2134     * protocols higher up. The order is:
2135     *
2136     *  - any isochronous events handled before any
2137     *    of the queues. We don't do that here, because
2138     *    we'll create the actual TD entries on demand.
2139     *  - The first queue is the "interrupt queue".
2140     *  - The second queue is the "control queue", split into low and high speed
2141     *  - The third queue is "bulk data".
2142     */
2143    static struct uhci *alloc_uhci(unsigned int io addr, unsigned int io_size)
2144    {
2145        int i, port;
```

```
2146        struct uhci *uhci;
2147        struct usb_bus *bus;
2148
2149        uhci = kmalloc(sizeof(*uhci), GFP_KERNEL);
2150        if (!uhci)
2151            return NULL;
2152
2153        memset(uhci, 0, sizeof(*uhci));
2154
2155        uhci->irq = -1;
2156        uhci->io_addr = io_addr;
2157        uhci->io_size = io_size;
2158
2159        spin_lock_init(&uhci->qh_remove_lock);
2160        INIT_LIST_HEAD(&uhci->qh_remove_list);
2161
2162        spin_lock_init(&uhci->urb_remove_lock);
2163        INIT_LIST_HEAD(&uhci->urb_remove_list);
2164
2165        nested_init(&uhci->urblist_lock);
2166        INIT_LIST_HEAD(&uhci->urb_list);
2167
2168        spin_lock_init(&uhci->framelist_lock);
2169
2170        /* We need exactly one page (per UHCI specs), how convenient */
2171        /* We assume that one page is atleast 4k (1024 frames * 4 bytes) */
2172        uhci->fl = (void *)__get_free_page(GFP_KERNEL);
2173        if (!uhci->fl)
2174            goto au_free_uhci;
2175
2176        bus = usb_alloc_bus(&uhci_device_operations);
2177        if (!bus)
2178            goto au_free_fl;
2179
2180        uhci->bus = bus;
2181        bus->hcpriv = uhci;
2182
```

首先为 uhci 数据结构分配空间，结构中 qh_remove_list、urb_remove_list、urb_list 等队列的作用以后会看到，这里是对这些队列头的初始化。然后就是为框架表分配空间，1024 个指针的大小为 4KB，正好是一个页面。前面讲过，uhci 数据结构代表着一个 USB 控制器，从而也代表着一条 USB 总线；然而，uhci 数据结构中的成分实际上还不足以全面地反映一条 USB 总线的状态，因此内核中又定义了一种 usb_bus 数据结构。所以，也要为 usb_bus 结构分配空间，并使 uhci 和 usb_bus 两个数据结构通过指针互相指向对方，连结成一个整体。这种数据结构定义于 include/linux/usb.h:

```
561    struct usb_bus {
```

```
562         int busnum;                     /* Bus number (in order of reg) */
563
564         struct usb_devmap devmap;       /* Device map */
565         struct usb_operations *op;      /* Operations (specific to the HC) */
566         struct usb_device *root_hub;    /* Root hub */
567         struct list_head bus_list;
568         void *hcpriv;                   /* Host Controller private data */
569
570         int bandwidth_allocated;        /* on this Host Controller; */
571                                         /* applies to Int. and Isoc. pipes; */
572                                         /* measured in microseconds/frame; */
573                                         /* range is 0..900, where 900 = */
574                                         /* 90% of a 1-millisecond frame */
575         int bandwidth_int_reqs;         /* number of Interrupt requesters */
576         int bandwidth_isoc_reqs;        /* number of Isoc. requesters */
577
578                                         /* usbdevfs inode list */
579         struct list_head inodes;
580     };
```

那么，为什么不把这些内容合在一起，而要分成两个数据结构呢？我们在前面讲过，USB 总线控制器有 UHCI 和 OHCI 两种界面，uhci 数据结构正是对 UHCI 界面的抽象。而 usb_bus 结构，则是对这两种界面的公共部分，即“USB 总线”层次上的抽象。正因为这样，usb_bus 结构中的指针 hcpriv 是个无类型(void)指针，它可以指向一个 uhci 数据结构，也可以指向一个 ohci 数据结构。

函数 usb_alloc_bus( )的作用是为 usb_bus 结构分配空间并进行初始化。由于简单，我们就不列出其代码了。这个数据结构中的指针 op 应该指向一个 usb_operations 数据结构，这里将其设置成指向 uhci_device_operations，分别定义于 include/linux/usb.h 和 drivers/usb/uhci.c：

```
550     struct usb_operations {
551         int (*allocate)(struct usb_device *);
552         int (*deallocate)(struct usb_device *);
553         int (*get_frame_number) (struct usb_device *usb_dev);
554         int (*submit_urb) (struct urb* purb);
555         int (*unlink_urb) (struct urb* purb);
556     };

1615    struct usb_operations uhci_device_operations = {
1616        uhci_alloc_dev,
1617        uhci_free_dev,
1618        uhci_get_current_frame_number,
1619        uhci_submit_urb,
1620        uhci_unlink_urb
1621    };
```

显然，这个数据结构为具体 USB 控制器界面的操作提供了函数指针。

我们继续看 alloc_uhci( )的代码(drivers/usb/uhci.c)。

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > alloc_uhci( )]

```
2183        /* Initialize the root hub */
2184
2185        /* UHCI specs says devices must have 2 ports, but goes on to say */
2186        /*  they may have more but give no way to determine how many they */
2187        /*  have. However, according to the UHCI spec, Bit 7 is always set */
2188        /*  to 1. So we try to use this to our advantage */
2189        for (port = 0; port < (io_size - 0x10) / 2; port++) {
2190            unsigned int portstatus;
2191
2192            portstatus = inw(io_addr + 0x10 + (port * 2));
2193            if (!(portstatus & 0x0080))
2194                break;
2195        }
2196        if (debug)
2197            info("detected %d ports", port);
2198
2199        /* This is experimental so anything less than 2 or greater than 8 is */
2200        /*  something weird and we'll ignore it */
2201        if (port < 2 || port > 8) {
2202            info("port count misdetected? forcing to 2 ports");
2203            port = 2;
2204        }
2205
2206        uhci->rh.numports = port;
2207
2208        /*
2209         * 9 Interrupt queues; link int2 to int1, int4 to int2, etc
2210         * then link int1 to control and control to bulk
2211         */
2212        for (i = 1; i < 9; i++) {
2213            struct uhci_td *td = &uhci->skeltd[i];
2214
2215            uhci_fill_td(td, 0,
                    (UHCI_NULL_DATA_SIZE << 21) | (0x7f << 8) | USB_PID_IN, 0);
2216            td->link = virt_to_bus(&uhci->skeltd[i - 1]);
2217        }
2218
2219
2220        uhci_fill_td(&uhci->skel_int1_td, 0,
                    (UHCI_NULL_DATA_SIZE << 21) | (0x7f << 8) | USB_PID_IN, 0);
2221        uhci->skel_int1_td.link =
                    virt_to_bus(&uhci->skel_ls_control_qh) | UHCI_PTR_QH;
2222
2223        uhci->skel_ls_control_qh.link =
                    virt_to_bus(&uhci->skel_hs_control_qh) | UHCI_PTR_QH;
```

```
2224        uhci->skel_ls_control_qh.element = UHCI_PTR_TERM;
2225
2226        uhci->skel_hs_control_qh.link =
                    virt_to_bus(&uhci->skel_bulk_qh) | UHCI_PTR_QH;
2227        uhci->skel_hs_control_qh.element = UHCI_PTR_TERM;
2228
2229        uhci->skel_bulk_qh.link =
                    virt_to_bus(&uhci->skel_term_qh) | UHCI_PTR_QH;
2230        uhci->skel_bulk_qh.element = UHCI_PTR_TERM;
2231
2232        /* This dummy TD is to work around a bug in Intel PIIX controllers */
2233        uhci_fill_td(&uhci->skel_term_td, 0,
                    (UHCI_NULL_DATA_SIZE << 21) | (0x7f << 8) | USB_PID_IN, 0);
2234        uhci->skel_term_td.link = UHCI_PTR_TERM;
2235
2236        uhci->skel_term_qh.link = UHCI_PTR_TERM;
2237        uhci->skel_term_qh.element = virt_to_bus(&uhci->skel_term_td);
```

USB 总线的根集中器总是与 USB 控制器集成在一起。对于 UHCI 界面的总线控制器，其 I/O 地址区间的前 16 个地址用于总线控制器本身，其余的就用于根集中器。根集中器的每个“端口”(port)占用两个地址。端口的数量则取决于具体的芯片，至少 2 个，最多 8 个。每个端口的状态寄存器中的 bit7 总是 1，所以代码中通过一个循环试读，以确定根集中器中端口的数量。

前面讲过，uhci 结构中的 skeltd[ ]用于 8 截中断交互队列，是个 uhci_td 数据结构的数组。结构名中的“td”是“交互描述结构”(transaction descriptor)的意思。这种数据结构定义于 drivers/usb/uhci.h:

```
175    struct uhci_td {
176        /* Hardware fields */
177          u32 link;
178        __u32 status;
179        __u32 info;
180        __u32 buffer;
181
182        /* Software fields */
183        unsigned int *frameptr;     /* Frame list pointer */
184        struct uhci_td *prevtd, *nexttd; /* Previous and next TD in queue */
185
186        struct usb_device *dev;
187        struct urb *urb;         /* URB this TD belongs to */
188
189        struct list_head list;
190    } __attribute__((aligned(16)));
```

每个 uhci_td 数据结构代表着一个交互请求，就好像是对 USB 控制器的一条指令。结构中前 4 个 32 位长字的作用是由 USB 控制器的硬件结构所确定了的，因而不能改变；不过硬件只认开头这 4 个字段，其余的字段则由软件定义和使用。数据结构的起点必须与 16 字节的边界对齐，这也是 USB 总线

控制器的硬件结构所要求的。这 4 个 32 位长字实际上相当于一组寄存器。在本书中，我们把由硬件使用的这一部分称为“交互描述块”，以区别于整个交互描述结构。结构头部的 4 个字段中，link 是指向下一个 uhci_td 数据结构的链接指针，buffer 则指向用于发送或接收的缓冲区，二者均为物理地址。此外，info 就好像是命令寄存器，相当于指令中的操作码。内核在 drivers/usb/uhci.c 中提供了一个 inline 函数 uhci_fill_td( )，用于设置 uhci_td 数据结构头部除 link 外的三个字段：

```
165    static void inline uhci_fill_td(struct uhci_td *td, __u32 status,
166            __u32 info, __u32 buffer)
167    {
168        td->status = status;
169        td->info = info;
170        td->buffer = buffer;
171    }
```

上面 2212 行的 for 循环是对中断交互队列的初始化。除第一个 uhci_td 数据结构 skeltd[0]以外，其余的 8 个 uhci_td 结构都设置成无数据的输入交互；并且都指向其前面的 uhci_td 数据结构；而 status 和 buffer 两个字段则都初始化成 0，表示“空操作”。

至于 skeltd[0]，则(2220～2221 行)初始化成指向低速 USB 设备的控制交互队列，即 uhci->skel_ls_control_qh。这里的 skel_int1_td、skel_ls_control_qh 都是 drivers/usb/uhci.h 中定义的宏定义：

```
235    #define UHCI_NUM_SKELTD      10
236    #define skel_int1_td         skeltd[0]
237    #define skel_int2_td         skeltd[1]
238    #define skel_int4_td         skeltd[2]
239    #define skel_int8_td         skeltd[3]
240    #define skel_int16_td        skeltd[4]
241    #define skel_int32_td        skeltd[5]
242    #define skel_int64_td        skeltd[6]
243    #define skel_int128_td       skeltd[7]
244    #define skel_int256_td       skeltd[8]
245    #define skel_term_td         skeltd[9]    /* To work around PIIX UHCI bug */
246
247    #define UHCI_NUM_SKELQH      4
248    #define skel_ls_control_qh   skelqh[0]
249    #define skel_hs_control_qh   skelqh[1]
250    #define skel_bulk_qh         skelqh[2]
251    #define skel_term_qh         skelqh[3]
```

这些宏定义实际上说明了各个队列的作用。例如，skel_ls_control_qh 表示 skelqh[0]是低速设备的控制传输队列头。那么，skel_int1_td 又表示什么呢？下面读者就会看到，它表示 skeltd[0]是在每一个时间框架中都会执行一遍的中断交互请求链入点。相应地，skel_int2_td 则表示 skeltd[1]是每两个框架才会执行一遍的中断交互请求链入点，等等。

与 skeltd[ ]不同，skelqh[ ]中的元素是 uhci_qh 数据结构，结构名中的“qh”表示这是个队列头(queue

head)。这种数据结构定义于 drivers/usb/uhci.h：

```
106    struct uhci_qh {
107        /* Hardware fields */
108        __u32 link;              /* Next queue */
109        __u32 element;               /* Queue element pointer */
110
111        /* Software fields */
112        /* Can't use list_head since we want a specific order */
113        struct usb_device *dev;          /* The owning device */
114
115        struct uhci_qh *prevqh, *nextqh;
116
117        struct list_head remove_list;
118    } __attribute__((aligned(16)));
```

这个数据结构也与 USB 控制器的硬件有关，其头部的 link 和 element 两个字段的作用是由硬件确定的，因而不能改变，其余的字段则只由软件使用。同样，我们有时把由硬件使用的这一部分称为“队列描述块”，以区别于整个数据结构。字段 link 实际上是指向下一个队列(描述块)的指针；elememt 也是指针，通常指向本队列中的第一个交互描述块，但是在特殊情况下也可以指向另一个队列描述块。二者均采用物理地址。由于 uhci_td 数据结构和 uhci_qh 数据结构都是与 16 字节边界对齐的，其地址的低 4 位一定是 0，所以 link 和 element 两个字段的低 4 位可以用作标志位。其中最低位为“结束”位 UHCI_PTR_TERM，当队列为空时，将队列头的指针 elememt 设置成 1(而不是 0)，就是因为其“结束”位为 1。此外，指针中还有一个标志位 UHCI_PTR_QH，用来告诉 USB 控制器所指向的是队列描述块还是交互描述块。从代码中可以看出，队列头 skel_ls_control_qh 通过其 link 字段指向 skel_hs_control_qh，而 skel_hs_control_qh 则又指向 skel_bulk_qh。所有这三个队列开始时都是空的。除这些以外，skeltd[ ]中还有个 skeltd[9]，即 skel_term_td，代码中的注释说这是因为 USB 总线控制器所在的 Intel PIIX 芯片内部有个错误，为了绕过这个错误而设的。还有，在 skelqh[ ]中还有个 skelqh[3]，即 skel_term_qh，这是为回收框架中尚未用完的时间而设置的，读者以后会看到其作用。

这样，对中断、控制和成块等三种交互请求队列的初始化就完成了。下面是对框架，即 1024 个等时交互请求队列的初始化。我们继续往下看(drivers/usb/uhci.c)。

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > alloc_uhci( )]

```
2238
2239        /*
2240         * Fill the frame list: make all entries point to
2241         * the proper interrupt queue.
2242         *
2243         * This is probably silly, but it's a simple way to
2244         * scatter the interrupt queues in a way that gives
2245         * us a reasonable dynamic range for irq latencies.
2246         */
```

```
2247        for (i = 0; i < 1024; i++) {
2248            struct uhci_td *irq = &uhci->skel_intl_td;
2249
2250            if (i & 1) {
2251                irq++;
2252                if (i & 2) {
2253                    irq++;
2254                    if (i & 4) {
2255                        irq++;
2256                        if (i & 8) {
2257                            irq++;
2258                            if (i & 16) {
2259                                irq++;
2260                                if (i & 32) {
2261                                    irq++;
2262                                    if (i & 64)
2263                                        irq++;
2264                                }
2265                            }
2266                        }
2267                    }
2268                }
2269            }
2270
2271            /* Only place we don't use the frame list routines */
2272            uhci->fl->frame[i] =  virt_to_bus(irq);
2273        }
2274
2275        return uhci;
2276
2277    /*
2278     * error exits:
2279     */
2280    au_free_fl:
2281        free_page((unsigned long)uhci->fl);
2282    au_free_uhci:
2283        kfree(uhci);
2284
2285        return NULL;
2286    }
```

框架表的 1024 个表项代表着 1024 个 1 毫秒框架，每个表项都是一个 uhci_td 结构指针，指向一个等时交互请求队列，而队列中的最后一个 uhci_td 结构则应该指向一个中断交互请求的 uhci_td 结构。然而，一开始时还没有等时交互请求，每个框架的等时交互请求队列都是空的，所以此时框架表的每个表项都应该直接指向中断交互请求队列中的某一个链入点、也就是 skeltd[ ]中的某个元素。可是，skeltd[ ]中共有 10 个 uhci_td 数据结构，应该指向哪一个呢？我们看代码，这里的 for 循环是很有意思

的。对于框架表中的每一个表项，先都假定指向 skeltd[0]，即 skel_int1_td（2248 行），让中断交互请求 skeltd[0]在每个框架中都会得到一次执行。然后，如果表项的框架号（即数组下标)为奇数(2250 行)，则把指针调整为指向 skeltd[1]。这样，skeltd[1] 就会每隔一个框架得到一次执行。同时，前面已经看到，skeltd[1]的指针 link 指向 skeltd[0]，所以 skeltd[0]还是在每个框架中都会得到执行，所以，skeltd[0]是 skel_int1_td，而 skeltd[1]则是 skel_int2_td。同理，如果框架号的最后两位为 3，则指向 skeltd[2]，使 skeltd[2]每隔 3 个框架就会得到一次执行，所以是 skel_int4_td。由于 skeltd[2]的指针 link 指向 skeltd[1]，所以 skeltd[1]和 skeltd[0]的执行保持不变，余类推。最后，如果框架号的最后 7 位为 127，则指向 skeltd[7]，使其每隔 127 个框架得到一次执行，所以是 skel_int128_td。这样，skeltd[ ]中的各个 uhci_td 结构就分别会在每 1 毫秒、2 毫秒、4 毫秒、… 、128 毫秒中得到一次执行。不过，我们在前面看到，skeltd[ ]中各个 uhci_td 结构的“操作码”均为 0，即“空操作”，因而实际上不会真的启动一次中断交互，而只是用来起到类似于队列头的作用。假定有个 USB 设备，需要每 16 毫秒对其启动一次中断交互，查询其状态变化，那就可以为之建立一个中断交互请求，并将其插入 skel_int16_td 与 skel_int8_td 之间，即 skeltd[4]和 skeltd[3]之间。理解了这段代码，就明白为什么说 skeltd[ ]是中断交互请求队列的“骨架”了。读者也许会问，为什么要用 uhci_td 结构来起“类似于队列头”的作用，而不是直接就用队列头，即 skelqh 数据结构呢？这一点后面会专门讲到。

至此，代表着 USB 总线的数据结构就初始化完毕了，我们回到前面 setup_uhci( )的代码中(2346 行)。接着，通过 reset_hc( )往 USB 总线控制器的命令寄存器中写入一个“全局总清”命令，以保证 USB 总线进入初始状态。这里的“hc”表示“主控制器”(host controller)。这个函数很简单，其代码在 drivers/usb/uhci.c 中：

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > reset_hc( )]

```
2087    static void reset_hc(struct uhci *uhci)
2088    {
2089        unsigned int io_addr = uhci->io_addr;
2090
2091        /* Global reset for 50ms */
2092        outw(USBCMD_GRESET, io addr + USBCMD);
2093        wait_ms(50);
2094        outw(0, io_addr + USBCMD);
2095        wait_ms(10);
2096    }
```

创建了代表着 USB 总线的 usb_bus 数据结构以后，还要通过 usb_register_bus( )向系统登记，其代码在 drivers/usb/usb.c 中：

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > usb_register_bus( )]

```
394     /**
395      *  usb register_bus - registers the USB host controller with the usb core
```

```
396      * @bus: pointer to the bus to register
397      *
398      */
399     void usb_register_bus(struct usb_bus *bus)
400     {
401         int busnum;
402 
403         busnum = find_next_zero_bit(busmap.busmap, USB_MAXBUS, 1);
404         if (busnum < USB_MAXBUS) {
405             set_bit(busnum, busmap.busmap);
406             bus->busnum = busnum;
407         } else
408             warn("too many buses");
409 
410         /* Add it to the list of buses */
411         list_add(&bus->bus_list, &usb_bus_list);
412 
413         usbdevfs_add_bus(bus);
414 
415         info("new USB bus registered, assigned bus number %d", bus->busnum);
416     }
```

内核中有个 usb_busmap 数据结构 busmap，里面只有一个成分，也叫 busmap，是用于 USB 总线的位图。此外，还有个队列 usb_bus_list。所谓登记就是从位图中分配一个标志位以及相应的总线号，并将代表着 USB 总线的 usb_bus 数据结构链入队列。

前面在 reset_hc( )中对 USB 总线进行了全局总清，接着还要通过 start_hc( )进一步设置 USB 控制器，这个函数的代码在 drivers/usb/uhci.c 中：

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > usb_register_bus( )]

```
2098    static void start_hc(struct uhci *uhci)
2099    {
2100        unsigned int io_addr = uhci->io_addr;
2101        int timeout = 1000;
2102 
2103        /*
2104         * Reset the HC - this will force us to get a
2105         * new notification of any already connected
2106         * ports due to the virtual disconnect that it
2107         * implies.
2108         */
2109        outw(USBCMD_HCRESET, io_addr + USBCMD);
2110        while (inw(io_addr + USBCMD) & USBCMD_HCRESET) {
2111            if (!--timeout) {
2112                printk(KERN_ERR "uhci: USBCMD_HCRESET timed out!\n");
```

```
2113                break;
2114            }
2115        }
2116
2117        /* Turn on all interrupts */
2118        outw(USBINTR_TIMEOUT | USBINTR_RESUME | USBINTR_IOC | USBINTR_SP,
2119            io_addr + USBINTR);
2120
2121        /* Start at frame 0 */
2122        outw(0, io_addr + USBFRNUM);
2123        outl(virt_to_bus(uhci->fl), io_addr + USBFLBASEADD);
2124
2125        /* Run and mark it configured with a 64-byte max packet */
2126        outw(USBCMD_RS | USBCMD_CF | USBCMD_MAXP, io_addr + USBCMD);
2127    }
```

这里一方面设置 USB 控制器中的中断控制寄存器，允许其在下列 4 种情况下向主机 CPU 发出中断请求：

(1) USBINTR_TIMEOUT。如果启动一次交互以后目标设备没有回应，从而造成超时。

(2) USBINTR_RESUME。为节约能源，如果一个 USB 设备(包括集中器)在一段时间内没有发生任何交互，就进入类似于“冬眠”的挂起(Suspend)模式，有设备从挂起模式回到正常运行模式时，可以引起一次中断。

(3) USBINTR_IOC。完成了一次交互(Interrupt-On-Completion)。如果某个交互请求表明要在完成以后引起中断，则 USB 控制器会在这个交互所在的框架结束之时发出中断请求。

(4) USBINTR_SP，从目标设备接收到的信包短于预期，称为“短信包”(short-packet)。

另一方面，还将控制器的“框架号寄存器”设成 0，使控制器从 0 号框架开始，并将框架表的基地址设置入“框架基地址寄存器”。最后，将“命令寄存器”中的“启动/停止”位 USBCMD_RS 设成 1，USB 控制器就开始运行了。当然，此时的 USB 总线在逻辑上还是空的，因而实际上还不会有交互，也不会有中断请求。此外，虽然总线控制器已开始扫描框架表，根集中器却尚未启用。

USB 控制器的中断服务程序是 uhci_interrupt( )，向系统的中断机制登记了中断服务程序(2358 行)以后（以及设置 USB 控制器的 PCI 配置寄存器组中的一个寄存器 USBLEGSUP 以后），便通过 uhci_start_root_hub( )启动根集中器的运行(drivers/usb/uhci.c)。

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > uhci_start_root_hub( )]

```
2307    int uhci_start_root_hub(struct uhci *uhci)
2308    {
2309        struct usb_device *dev;
2310
2311        dev = usb_alloc_dev(NULL, uhci->bus);
2312        if (!dev)
2313            return -1;
2314
```

```
2315        uhci->bus->root_hub = dev;
2316        usb_connect(dev);
2317
2318        if (usb_new_device(dev) != 0) {
2319            usb_free_dev(dev);
2320
2321            return -1;
2322        }
2323
2324        return 0;
2325    }
```

连接在 USB 总线上的每个设备、包括根集中器、都需要有个 usb_device 数据结构作为代表，这是对 USB 设备的抽象，定义于 include\linux\usb.h 中：

```
584    struct usb_device {
585        int devnum;                 /* Device number on USB bus */
586        int slow;                   /* Slow device? */
587
588        atomic_t refcnt;            /* Reference count */
589
590        unsigned int toggle[2]; /* one bit for each endpoint ([0] = IN, [1] = OUT) */
591        unsigned int halted[2];
                           /* endpoint halts; one bit per endpoint # & direction; */
592                        /* [0] = IN, [1] = OUT */
593        int epmaxpacketin[16];      /* INput endpoint specific maximums */
594        int epmaxpacketout[16];     /* OUTput endpoint specific maximums */
595
596        struct usb_device *parent;
597        struct usb_bus *bus;        /* Bus we're part of */
598
599        struct usb_device_descriptor descriptor;/* Descriptor */
600        struct usb_config_descriptor *config;   /* All of the configs */
601        struct usb_config_descriptor *actconfig;/* the active configuration */
602
603        char **rawdescriptors;      /* Raw descriptors for each config */
604
605        int have_langid;            /* whether string_langid is valid yet */
606        int string_langid;          /* language ID for strings */
607
608        void *hcpriv;               /* Host Controller private data */
609
610                                    /* usbdevfs inode list */
611        struct list_head inodes;
612        struct list_head filelist;
613
614        /*
```

```
615          * Child devices - these can be either new devices
616          * (if this is a hub device), or different instances
617          * of this same device.
618          *
619          * Each instance needs its own set of data structures.
620          */
621
622         int maxchild;                 /* Number of ports if hub */
623         struct usb_device *children[USB_MAXCHILDREN];
624     };
```

USB 总线上的每个设备都有个动态分配的设备号，由于在一条 USB 总线上的设备不能超过 127 个，所以设备号的范围为 1～127。0 表示尚未分配。如果具体的设备是个集中器，则通过一个指针数组 children[ ]指向连接在该集中器上的设备(usb_device 结构)。而每个设备(除根集中器之外)，则通过指针 parent 指向其所连接的集中器(usb_device 结构)，同时又通过指针 bus 指向其所在 USB 总线的 usb_bus 数据结构。这样，最终所有的 usb_device 结构就会连接成一个树状的结构，反映出一条 USB 总线的拓扑图形。此外，usb_device 结构中有个重要的成分 descriptor，是个 usb_device_descriptor 数据结构，定义于 include/linux/usb.h 中：

```
218     /* Device descriptor */
219     struct usb_device_descriptor {
220             __u8  bLength;
221             __u8  bDescriptorType;
222             __u16 bcdUSB;
223             __u8  bDeviceClass;
224             __u8  bDeviceSubClass;
225             __u8  bDeviceProtocol;
226             __u8  bMaxPacketSize0;
227             __u16 idVendor;
228             __u16 idProduct;
229             __u16 bcdDevice;
230             __u8  iManufacturer;
231             __u8  iProduct;
232             __u8  iSerialNumber;
233             __u8  bNumConfigurations;
234     } __attribute__ ((packed));
```

这个数据结构的内容是由 USB 设备的硬件提供的，其内容固化在设备的硬件中，可以通过一次控制交互从设备读入。显然，其内容与 PCI 设备配置寄存器组中的一些信息相似。不过，USB 控制器在 PCI 总线上，是 PCI 设备；而 USB 设备则直接或间接地(通过集中器)连接在 USB 控制器上，其本身并非 PCI 设备。至于 usb_device 结构中的其他成分，则读者随着代码的阅读自会明白。

像 USB 总线上的其他集中器一样，根集中器本身也是个 USB 设备。所以要为其分配一个 usb_device 结构，并让相应 usb_bus 结构中的指针 root_hub 指向这个数据结构。

然后，还要通过 usb_connect( )为这个设备间动态地分配一个 USB 总线设备号（drivers/usb/usb.c)。

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > uhci_start_root_hub( ) > usb_connect( )]

```
1667    /*
1668     * Connect a new USB device. This basically just initializes
1669     * the USB device information and sets up the topology - it's
1670     * up to the low-level driver to reset the port and actually
1671     * do the setup (the upper levels don't know how to do that).
1672     */
1673    void usb_connect(struct usb_device *dev)
1674    {
1675        int devnum;
1676        // FIXME needs locking for SMP!!
1677        /* why? this is called only from the hub thread,
1678         * which hopefully doesn't run on multiple CPU's simultaneously 8-)
1679         */
1680        dev->descriptor.bMaxPacketSize0 = 8;  /* Start off at 8 bytes  */
1681    #ifndef DEVNUM_ROUND_ROBIN
1682        devnum = find_next_zero_bit(dev->bus->devmap.devicemap, 128, 1);
1683    #else   /* round robin alloc of devnums */
1684        /* Try to allocate the next devnum beginning at devnum_next. */
1685        devnum = find_next_zero_bit(dev->bus->devmap.devicemap, 128, devnum_next);
1686        if (devnum >= 128)
1687            devnum = find_next_zero_bit(dev->bus->devmap.devicemap, 128, 1);
1688
1689        devnum_next = devnum + 1;
1690        if (devnum_next >= 128)
1691            devnum_next = 1;
1692    #endif  /* round_robin alloc of devnums */
1693
1694        if (devnum < 128) {
1695            set_bit(devnum, dev->bus->devmap.devicemap);
1696            dev->devnum = devnum;
1697        }
1698    }
```

接下去，就要与目标设备(这里是根集中器)建立起实际的联系了，这就是对目标设备的“枚举”。它是一个颇为复杂的过程。一般而言，当把一个 USB 设备插入一个 USB 集中器的某个“端口”时，集中器就会检测到设备的接入，从而在下一次受到主机通过中断交互查询时就会向其报告。集中器的端口在没有设备连接时都处于关闭状态，插入设备以后也不会自动打开，必须由主机通过控制交互发出命令予以打开。所以，得到集中器的报告以后，主机中的 USB 驱动程序就会为新插上的设备调度若干个控制交互，并向集中器发出打开这个端口的命令。这样，新插入的设备就出现在 USB 总线上了。要在主控制器与 USB 设备之间通信，USB 设备就必须有个(在总线上)惟一的地址。所有的 USB 设备在一开始时都使用“默认”地址 0，与主机中的 USB 控制器建立起初始通信以后再由主机(当然，实际上是驱动软件)指定一个地址，以后就一直使用这个指定的地址，直到断开与总线的连接为止。这里，读

者很自然会产生一个问题：如果连接在总线上的所有 USB 设备在一开始时的地址都是 0，那主机的 USB 控制器在初始化时期怎么能与具体的目标设备通信呢？其实很简单，不管有多少个 USB 设备同时连接到各个集中器上，总线控制器总是打开一个端口就指定一个地址，然后再打开下一个端口，决不会在尚未为已打开的端口所连接的设备指定好地址之前就又打开另一个端口。这样，在同一时间里，USB 总线上最多只会有一个设备在使用地址 0。

枚举过程中主机与设备间的信息交换不仅仅是为设备指定地址，总的来说有下面这一些步骤：

(1)　为设备指定地址。

(2)　从设备读入其 usb_device_descriptor 数据结构。

(3)　从设备读入其所有的“配置”描述结构。

(4)　选择或改变设备的配置。

根设备与总线控制器的连接是固定的，不需要通过另一个集中器的报告，所以直接就可以开始其枚举过程，这是由 usb_new_device( )完成的，其代码在 drivers/usb/usb.c 中。我们分段阅读。

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > uhci_start_root_hub( ) > usb_new_device( )]

```
2079    /*
2080     * By the time we get here, the device has gotten a new device ID
2081     * and is in the default state. We need to identify the thing and
2082     * get the ball rolling..
2083     *
2084     * Returns 0 for success, != 0 for error.
2085     */
2086    int usb_new_device(struct usb_device *dev)
2087    {
2088        int err;
2089
2090        /* USB v1.1 5.5.3 */
2091        /* We read the first 8 bytes from the device descriptor to get to */
2092        /*  the bMaxPacketSize0 field. Then we set the maximum packet size */
2093        /*  for the control pipe, and retrieve the rest */
2094        dev->epmaxpacketin [0] = 8;
2095        dev->epmaxpacketout[0] = 8;
2096
2097        err = usb_set_address(dev);
2098        if (err < 0) {
2099            err("USB device not accepting new address=%d (error=%d)",
2100                dev->devnum, err);
2101            clear_bit(dev->devnum, &dev->bus->devmap.devicemap);
2102            dev->devnum = -1;
2103            return 1;
2104        }
2105
2106        wait_ms(10);    /* Let the SET_ADDRESS settle */
2107
```

```
2108        err = usb_get_descriptor(dev, USB_DT_DEVICE, 0, &dev->descriptor, 8);
2109        if (err < 8) {
2110            if (err < 0)
2111                err("USB device not responding, giving up (error=%d)", err);
2112            else
2113                err("USB device descriptor short read (expected %i, got %i)", 8, err);
2114            clear_bit(dev->devnum, &dev->bus->devmap.devicemap);
2115            dev->devnum = -1;
2116            return 1;
2117        }
2118        dev->epmaxpacketin [0] = dev->descriptor.bMaxPacketSize0;
2119        dev->epmaxpacketout[0] = dev->descriptor.bMaxPacketSize0;
2120
2121        err = usb_get_device_descriptor(dev);
2122        if (err < sizeof(dev->descriptor)) {
2123            if (err < 0)
2124                err("unable to get device descriptor (error=%d)", err);
2125            else
2126                err("USB device descriptor short read (expected %i, got %i)",
2127                    sizeof(dev->descriptor), err);
2128
2129            clear_bit(dev->devnum, &dev->bus->devmap.devicemap);
2130            dev->devnum = -1;
2131            return 1;
2132        }
2133
```

USB 设备内部有若干用于与主机通信的“端点”(endpoint)，每个端点都有个端点号，不同类型的传输使用不同的端点。每一个 USB 设备至少要提供用于控制传输的控制端点，其端点号为 0。其他端点则视具体设备而定，例如扫描器就不会有等时传输端点，因为不需要。控制端点是双向的，既可用于输入传输，也可用于输出传输。在每次传输中，通信的一方总是主机中的 USB 主控制器，而另一方就由设备号和端点号惟一地确定，称为一个“管道”(pipe)。这里，首先将把目标设备（在这里是根集中器）的控制端点两个方向上的信包大小都假定为 8，然后就通过 usb_set_address( )为目标设备指定地址(drivers/usb/usb.c)。

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > uhci_start_root_hub( ) > usb_new_device( ) > usb_set_address( )]

```
1708    int usb_set_address(struct usb_device *dev)
1709    {
1710        return usb_control_msg(dev, usb_snddefctrl(dev), USB_REQ_SET_ADDRESS,
1711            0, dev->devnum, 0, NULL, 0, HZ * GET_TIMEOUT);
1712    }
```

这里的 usb_control_msg( )专门用来调度控制传输、并等待传输的完成。我们将在后面讲述具体 USB 设备的操作时详细介绍这个函数，现在暂且假定调用这个函数就可以让 USB 控制器把一个控制报文发

送给目标设备。如果控制传输顺利完成就返回 0，否则返回一个负的出错代码。调用的参数表明，使用的管道是“snddefctrl”，即默认的地址 0 加上控制端点号 0，交互的方向为输出。此外，发送给设备的命令码是 USB_REQ_SET_ADDRESS，即设置地址；而所设置的地址为 dev->devnum，即目标设备的设备号；没有附加数据；允许等待传输完成的时间是 3 秒，即 HZ*GET_TIMEOUT，这里 HZ 表示一秒，而 GET_TIMEOUT 则定义为 3。

假定目标设备顺利地完成了设置地址的操作，经过一个短暂的延迟(2106 行)以后，就可以使用新的地址启动另一次控制传输，从目标设备读入其设备描述块，即 usb_device_descriptor 数据结构了。这是由 usb_get_descriptor( )完成的，其代码在 drivers/usb/usb.c 中：

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > uhci_start_root_hub( ) > usb_new_device( ) > usb_get_descriptor ( )]

```
1714    int usb_get_descriptor(struct usb_device *dev, unsigned char type,
                                        unsigned char index, void *buf, int size)
1715    {
1716        int i = 5;
1717        int result;
1718
1719        memset(buf,0,size); // Make sure we parse really received data
1720
1721        while (i--) {
1722            if ((result = usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
1723                USB_REQ_GET_DESCRIPTOR, USB_DIR_IN,
1724                (type << 8) + index, 0, buf, size, HZ * GET_TIMEOUT)) > 0 ||
1725                 result == -EPIPE)
1726                break;  /* retry if the returned length was 0; flaky device */
1727        }
1728        return result;
1729    }
```

这一次，使用的管道为“rcvctrlpipe”，即目标设备的地址加端点号 0，方向为输入。对设备的命令为 USB_REQ_GET_DESCRIPTOR，即读取描述块。可是，设备中不仅有“设备描述块”，还有“接口描述块”、“配置描述块”等等，所以前面在调用时指明为 USB_DT_DEVICE，即设备描述块。有关的几个常数定义于 include/linux/usb.h：

```
44    /*
45     * Descriptor types
46     */
47    #define USB_DT_DEVICE            0x01
48    #define USB_DT_CONFIG            0x02
49    #define USB_DT_STRING            0x03
50    #define USB_DT_INTERFACE         0x04
51    #define USB_DT_ENDPOINT          0x05
```

调用参数还表明，只从设备描述结构中读取 8 字节。完成了传输、将所需的内容读入 dev->descriptor

以后，还要根据来自设备的数据，调整控制交互信包的最大容量（2118～2119 行）。然后，再通过 usb_get_device_descriptor( )重读一次目标设备的设备描述结构。其代码在 drivers/usb/usb.c 中。

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > uhci_start_root_hub( ) > usb_new_device( ) > usb_get_device_descriptor( )]

```
1746    int usb_get_device_descriptor(struct usb_device *dev)
1747    {
1748        int ret = usb_get_descriptor(dev, USB_DT_DEVICE, 0, &dev->descriptor,
1749                         sizeof(dev->descriptor));
1750        if (ret >= 0) {
1751            le16_to_cpus(&dev->descriptor.bcdUSB);
1752            le16_to_cpus(&dev->descriptor.idVendor);
1753            le16_to_cpus(&dev->descriptor.idProduct);
1754            le16_to_cpus(&dev->descriptor.bcdDevice);
1755        }
1756        return ret;
1757    }
```

这一次读入的是整个设备描述结构。那么，为什么先要按 8 个字节先读一次呢？这是因为开始时还不知道对方所支持的信包容量，这个信息在对方的设备描述结构的开头 8 个字节中；然而，8 字节的信包容量是所有设备都支持的，所以先按最低标准先读一次，读入了设备描述结构的开头 8 个字节以后，就知道了对方所支持的最大信包容量，以后就可以按这个容量传输了。此外，从设备读入的 16 位整数都是“little ending”的格式，所以要把它们转换成主机 CPU 所采用的格式。

读入设备描述结构以后，接着还要读入有关设备配置的信息。我们继续往下读 usb_new_device( ) 的代码(drivers/usb/usb.c)。

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > uhci_start_root_hub( ) > usb_new_device( )]

```
2134        err = usb_get_configuration(dev);
2135        if (err < 0) {
2136            err("unable to get device %d configuration (error=%d)",
2137                dev->devnum, err);
2138            clear_bit(dev->devnum, &dev->bus->devmap.devicemap);
2139            dev->devnum = -1;
2140            usb_free_dev(dev);
2141            return 1;
2142        }
2143
2144        /* we set the default configuration here */
2145        err = usb_set_configuration(dev, dev->config[0].bConfigurationValue);
2146        if (err) {
2147            err("failed to set device %d default configuration (error=%d)",
2148                dev->devnum, err);
```

```
2149            clear_bit(dev->devnum, &dev->bus->devmap.devicemap);
2150            dev->devnum = -1;
2151            return 1;
2152        }
2153
2154        dbg("new device strings: Mfr=%d, Product=%d, SerialNumber=%d",
2155            dev->descriptor.iManufacturer, dev->descriptor.iProduct,
                                                 dev->descriptor.iSerialNumber);
2156    #ifdef DEBUG
2157        if (dev->descriptor.iManufacturer)
2158            usb_show_string(dev, "Manufacturer", dev->descriptor.iManufacturer);
2159        if (dev->descriptor.iProduct)
2160            usb_show_string(dev, "Product", dev->descriptor.iProduct);
2161        if (dev->descriptor.iSerialNumber)
2162            usb_show_string(dev, "SerialNumber", dev->descriptor.iSerialNumber);
2163    #endif
2164
2165        /* now that the basic setup is over, add a /proc/bus/usb entry */
2166        usbdevfs_add_device(dev);
2167
2168        /* find drivers willing to handle this device */
2169        usb_find_drivers(dev);
2170
2171        /* userspace may load modules and/or configure further */
2172        call_policy ("add", dev);
2173
2174        return 0;
2175    }
```

设备的每一种具体的配置都由一个 usb_config_descriptor 数据结构加以描述，定义于include/linux/usb.h：

```
280     /* Configuration descriptor information.. */
281     struct usb_config_descriptor {
282         __u8  bLength           __attribute__ ((packed));
283         __u8  bDescriptorType   __attribute__ ((packed));
284         __u16 wTotalLength __attribute__ ((packed));
285         __u8  bNumInterfaces    __attribute__ ((packed));
286         __u8  bConfigurationValue __attribute__ ((packed));
287         __u8  iConfiguration    __attribute__ ((packed));
288         __u8  bmAttributes  __attribute__ ((packed));
289         __u8  MaxPower          __attribute__ ((packed));
290
291         struct usb_interface *interface;
292
293         unsigned char *extra;   /* Extra descriptors */
294         int extralen;
```

```
295     };
```

结构中指针 interface 以前的部分(282～289 行)与直接从设备读取的“配置描述块”格式相同。函数 usb_get_configuration( )的代码在 drivers/usb/usb.c 中：

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > uhci_start_root_hub( ) > usb_new_device( ) > usb_get_configuration( )]

```
1926    int usb_get_configuration(struct usb_device *dev)
1927    {
1928        int result;
1929        unsigned int cfgno, length;
1930        unsigned char buffer[8];
1931        unsigned char *bigbuffer;
1932        struct usb_config_descriptor *desc =
1933            (struct usb_config_descriptor *)buffer;
1934
1935        if (dev->descriptor.bNumConfigurations > USB_MAXCONFIG) {
1936            warn("too many configurations");
1937            return -EINVAL;
1938        }
1939
1940        if (dev->descriptor.bNumConfigurations < 1) {
1941            warn("not enough configurations");
1942            return -EINVAL;
1943        }
1944
1945        dev->config = (struct usb config_descriptor *)
1946            kmalloc(dev->descriptor.bNumConfigurations *
1947            sizeof(struct usb config descriptor), GFP_KERNEL);
1948        if (!dev->config) {
1949            err("out of memory");
1950            return -ENOMEM;
1951        }
1952        memset(dev->config, 0, dev->descriptor.bNumConfigurations *
1953            sizeof(struct usb_config_descriptor));
1954
1955        dev->rawdescriptors = (char **)kmalloc(sizeof(char *) *
1956            dev->descriptor.bNumConfigurations, GFP_KERNEL);
1957        if (!dev->rawdescriptors) {
1958            err("out of memory");
1959            return -ENOMEM;
1960        }
1961
1962        for (cfgno = 0; cfgno < dev->descriptor.bNumConfigurations; cfgno++) {
1963            /* We grab the first 8 bytes so we know how long the whole */
1964            /*  configuration is */
```

```
1965            result = usb_get_descriptor(dev, USB_DT_CONFIG, cfgno, buffer, 8);
1966            if (result < 8) {
1967                if (result < 0)
1968                    err("unable to get descriptor");
1969                else {
1970                    err("config descriptor too short (expected %i, got %i)",
                                                          8, result);
1971                    result = -EINVAL;
1972                }
1973                goto err;
1974            }
1975
1976            /* Get the full buffer */
1977            length = le16_to_cpu(desc->wTotalLength);
1978
1979            bigbuffer = kmalloc(length, GFP_KERNEL);
1980            if (!bigbuffer) {
1981                err("unable to allocate memory for configuration descriptors");
1982                result = -ENOMEM;
1983                goto err;
1984            }
1985
1986            /* Now that we know the length, get the whole thing */
1987            result = usb_get_descriptor(dev, USB_DT_CONFIG, cfgno, bigbuffer, length);
1988            if (result < 0) {
1989                err("couldn't get all of config descriptors");
1990                kfree(bigbuffer);
1991                goto err;
1992            }
1993
1994            if (result < length) {
1995                err("config descriptor too short (expected %i, got %i)", length, result);
1996                result = -EINVAL;
1997                kfree(bigbuffer);
1998                goto err;
1999            }
2000
2001            dev->rawdescriptors[cfgno] = bigbuffer;
2002
2003            result = usb_parse_configuration(dev, &dev->config[cfgno], bigbuffer);
2004            if (result > 0)
2005                dbg("descriptor data left");
2006            else if (result < 0) {
2007                result = -EINVAL;
2008                goto err;
2009            }
2010        }
2011
```

```
2012        return 0;
2013    err:
2014        dev->descriptor.bNumConfigurations = cfgno;
2015        return result;
2016    }
```

每个 USB 设备至少有一个配置描述块。设备描述块中的字段 bNumConfigurations 说明了本设备有几个配置描述块，但最多不能超过 USB_MAXCONFIG，即 8 个。根据这个字段的数值，可以为目标设备分配用于相应描述结构的空间，并让目标设备的 usb_device 数据结构通过其指针 config 指向这块空间。此外，usb_device 数据结构中还有个指针 rawdescriptors，应该指向一个指针数组，该数组中的每一个元素都指向一个从设备读入的配置描述块，所以其大小也取决于配置描述块的个数。做好了这些准备以后，就通过一个 for 循环（1962 行）从设备依次读入各个配置描述块。配置描述块的读入还是由 usb_get_descriptor( )完成，但是从代码中可以看出现在要读入的是 USB_DT_CONFIG。每次先读入 8 个字节，读入的信息暂时存放在 buffer 中。根据读入信息中的 wTotalLength 可以知道实际的大小，然后再分配足够大的空间，再调用一次 usb_get_descriptor( )，把整个描述块读进来，并将其起始地址放在 rawdescriptors 所指的指针数组中。

每个配置描述块是作为一个整体读入缓冲区的。配置描述块中包含着若干个"接口描述块"，而每个接口描述块中又可以包含若干个"端点描述块"，各种次层描述块的数量则因具体的配置而异，所以配置描述块的大小并非常数。同时，这些描述块中又可以包含一些由具体设备的制造商或行业协会自行定义的次层描述块。所以，每读入一个配置描述块以后，都要通过 usb_parse_configuration()加以分析辨认，从配置描述块中分解出各个次层描述块，并为这些描述块建立起相应的数据结构，以形成对目标设备各个层次的描述。其代码在 drivers/usb/usb.c 中：

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > uhci_start_root_hub( ) > usb_new_device( ) > usb_get_configuration( ) > usb_parse_configuration( )]

```
1399    int usb_parse_configuration(struct usb_device *dev,
                            struct usb_config_descriptor *config, char *buffer)
1400    {
1401        int i, retval, size;
1402        struct usb_descriptor_header *header;
1403
1404        memcpy(config, buffer, USB_DT_CONFIG_SIZE);
1405        le16_to_cpus(&config->wTotalLength);
1406        size = config->wTotalLength;
1407
1408        if (config->bNumInterfaces > USB_MAXINTERFACES) {
1409            warn("too many interfaces");
1410            return -1;
1411        }
1412
1413        config->interface = (struct usb_interface *)
1414            kmalloc(config->bNumInterfaces *
1415            sizeof(struct usb_interface), GFP_KERNEL);
```

```
1416        dbg("kmalloc IF %p, numif %i", config->interface, config->bNumInterfaces);
1417        if (!config->interface) {
1418            err("out of memory");
1419            return -1;
1420        }
1421
1422        memset(config->interface, 0,
1423               config->bNumInterfaces * sizeof(struct usb_interface));
1424
1425        buffer += config->bLength;
1426        size -= config->bLength;
1427
1428        for (i = 0; i < config->bNumInterfaces; i++) {
1429            int numskipped, len;
1430            char *begin;
1431
1432            /* Skip over the rest of the Class Specific or Vendor */
1433            /*  Specific descriptors */
1434            begin = buffer;
1435            numskipped = 0;
1436            while (size >= sizeof(struct usb_descriptor_header)) {
1437                header = (struct usb_descriptor_header *)buffer;
1438
1439                if ((header->bLength > size) || (header->bLength < 2)) {
1440                    err("invalid descriptor length of %d", header->bLength);
1441                    return -1;
1442                }
1443
1444                /* If we find another descriptor which is at or below */
1445                /*  us in the descriptor heirarchy then we're done  */
1446                if ((header->bDescriptorType == USB_DT_ENDPOINT) ||
1447                    (header->bDescriptorType == USB_DT_INTERFACE) ||
1448                    (header->bDescriptorType == USB_DT_CONFIG) ||
1449                    (header->bDescriptorType == USB_DT_DEVICE))
1450                    break;
1451
1452                dbg("skipping descriptor 0x%X", header->bDescriptorType);
1453                numskipped++;
1454
1455                buffer += header->bLength;
1456                size -= header->bLength;
1457            }
1458            if (numskipped)
1459                dbg("skipped %d class/vendor specific endpoint descriptors",
                                                                   numskipped);
1460
1461            /* Copy any unknown descriptors into a storage area for */
1462            /*  drivers to later parse */
```

```
1463            len = (int)(buffer - begin);
1464            if (!len) {
1465                config->extra = NULL;
1466                config->extralen = 0;
1467            } else {
1468                config->extra = kmalloc(len, GFP_KERNEL);
1469                if (!config->extra) {
1470                    err("couldn't allocate memory for config extra descriptors");
1471                    config->extralen = 0;
1472                    return -1;
1473                }
1474
1475                memcpy(config->extra, begin, len);
1476                config->extralen = len;
1477            }
1478
1479            retval = usb_parse_interface(dev, config->interface + i, buffer, size);
1480            if (retval < 0)
1481                return retval;
1482
1483            buffer += retval;
1484            size -= retval;
1485        }
1486
1487        return size;
1488    }
```

参数 config 指向一个 usb_config_descriptor 数据结构，而 buffer 则指向刚从设备读入的“原始”配置描述块缓冲区。首先将缓冲区的前 9 个字节复制到 usb_config_descriptor 结构中，这里的 USB_DT_CONFIG_SIZE 定义为 9。配置描述块中的 bNumInterfaces 说明在本配置中设备中有几个“接口”。所谓一个接口，就是指 USB 设备中的一项特定的功能，也就是逻辑设备。我们可以在逻辑上把每个具体的“接口”看成一组端点的集合，就好像把传统的外设接口看成一组寄存器或存储区间的集合一样。但是，即使是对于同一个接口，即同一个逻辑设备，也还是可以有几种不同的“设置”可供选用，在某种意义上可以看作是同一类逻辑设备的不同实现。这就好像同样是录音机却仍可以组合成不同的性能，将各个按扭用于不同的控制对像。当然，在任何特定的时刻，只能选用其中一种。所以，在内存中为每一个逻辑设备建立起一个 usb_interface 数据结构。这种数据结构定义于 include/linux/usb.h：

```
269    struct usb_interface {
270        struct usb_interface_descriptor *altsetting;
271
272        int act_altsetting;     /* active alternate setting */
273        int num_altsetting;     /* number of alternate settings */
274        int max_altsetting;             /* total memory allocated */
275
```

```
276         struct usb_driver *driver;  /* driver */
277         void *private_data;
278     };
```

结构中的指针 altsetting 指向一个 usb_interface_descriptor 数据结构的数组，数组中的每个元素都是来自设备的“接口描述块”，代表着该接口的一个具体“设置”，而 act_altsetting 则说明当前选用的是其中的哪一个。这个数据结构是在 include/linux/usb.h 中定义的：

```
251     /* Interface descriptor */
252     struct usb_interface_descriptor {
253         __u8  bLength        __attribute__  ((packed));
254         __u8  bDescriptorType   __attribute__  ((packed));
255         __u8  bInterfaceNumber  __attribute__  ((packed));
256         __u8  bAlternateSetting __attribute__  ((packed));
257         __u8  bNumEndpoints __attribute__  ((packed));
258         __u8  bInterfaceClass   __attribute__  ((packed));
259         __u8  bInterfaceSubClass __attribute__  ((packed));
260         __u8  bInterfaceProtocol __attribute__  ((packed));
261         __u8  iInterface     __attribute__  ((packed));
262
263         struct usb_endpoint_descriptor *endpoint;
264
265         unsigned char *extra;   /* Extra descriptors */
266         int extralen;
267     };
```

指针 endpoint 以前的字段均来自目标设备。注意这个结构中的 bInterfaceNumber 和 bAlternateSetting 两个字段。当一个接口有几个不同的“设置”可供选用时，设备就会提供相应数量的接口描述块，这些描述块都有相同的“接口号”，但是“设置号”却各不相同，其中第一个接口描述块的“设置号”为 0。所以，在一连串的接口描述块中，每个“设置号”为 0 的描述块都标志着一个接口的开始。一个接口至少包含一个设置。在配置描绘块头部中说明了本配置有几个接口，但是这并不说明有几个接口描述块，所以才需要“parse”，即分析辨认。

知道了目标设备中有几个接口，就可以为这些接口的 usb_interface 结构分配足够的空间(1413 行)，然后通过一个 for 循环(1428 行)，依次从已经读入缓冲区的信息中找到属于各个接口的描述块。找到了一个接口描述的开头以后，就通过 usb_parse_interface( )进一步分析和辨认出这个接口的所有描述块，其代码也在 drivers/usb/usb.c 中。这个函数比较长，我们得要分段阅读。

[pci_module_init( ) > pci_register_driver( ) > pci_announce_device( ) > uhci_pci_probe( ) > setup_uhci( ) > uhci_start_root_hub( ) > usb_new_device( ) > usb_get_configuration( ) > usb_parse_configuration( ) > usb_parse_interface( )]

```
1247    static int usb_parse_interface(struct usb_device *dev,
                struct usb_interface *interface, unsigned char *buffer, int size)
1248    {
```

```
1249        int i, len, numskipped, retval, parsed = 0;
1250        struct usb_descriptor_header *header;
1251        struct usb_interface_descriptor *ifp;
1252        unsigned char *begin;
1253
1254        interface->act_altsetting = 0;
1255        interface->num_altsetting = 0;
1256        interface->max_altsetting = USB_ALTSETTINGALLOC;
1257
1258        interface->altsetting =
     kmalloc(sizeof(struct usb_interface_descriptor) * interface->max_altsetting,
                                                                          GFP_KERNEL);
1259
1260        if (!interface->altsetting) {
1261            err("couldn't kmalloc interface->altsetting");
1262            return -1;
1263        }
1264
1265        while (size > 0) {
1266            if (interface->num_altsetting >= interface->max_altsetting) {
1267                void *ptr;
1268                int oldmas;
1269
1270                oldmas = interface->max_altsetting;
1271                interface->max_altsetting += USB_ALTSETTINGALLOC;
1272                if (interface->max_altsetting > USB_MAXALTSETTING) {
1273                    warn("too many alternate settings (max %d)",
1274                        USB_MAXALTSETTING);
1275                    return -1;
1276                }
1277
1278                ptr = interface->altsetting;
1279                interface->altsetting = kmalloc(
            sizeof(struct usb_interface_descriptor) * interface->max_altsetting,
            GFP_KERNEL);
1280                if (!interface->altsetting) {
1281                    err("couldn't kmalloc interface->altsetting");
1282                    interface->altsetting = ptr;
1283                    return -1;
1284                }
1285                memcpy(interface->altsetting, ptr,
                        sizeof(struct usb_interface_descriptor) * oldmas);
1286
1287                kfree(ptr);
1288            }
1289
1290            ifp = interface->altsetting + interface->num_altsetting;
1291            interface->num_altsetting++;
```

```
1292
1293            memcpy(ifp, buffer, USB_DT_INTERFACE_SIZE);
1294
1295            /* Skip over the interface */
1296            buffer += ifp->bLength;
1297            parsed += ifp->bLength;
1298            size -= ifp->bLength;
1299
1300            begin = buffer;
1301            numskipped = 0;
1302
1303            /* Skip over any interface, class or vendor descriptors */
1304            while (size >= sizeof(struct usb_descriptor_header)) {
1305                header = (struct usb_descriptor_header *)buffer;
1306
1307                if (header->bLength < 2) {
1308                    err("invalid descriptor length of %d", header->bLength);
1309                    return -1;
1310                }
1311
1312                /* If we find another descriptor which is at or below */
1313                /*  us in the descriptor heirarchy then return */
1314                if ((header->bDescriptorType == USB_DT_INTERFACE) ||
1315                    (header->bDescriptorType == USB_DT_ENDPOINT) ||
1316                    (header->bDescriptorType == USB_DT_CONFIG) ||
1317                    (header->bDescriptorType == USB_DT_DEVICE))
1318                    break;
1319
1320                numskipped++;
1321
1322                buffer += header->bLength;
1323                parsed += header->bLength;
1324                size -= header->bLength;
1325            }
1326
1327            if (numskipped)
1328                dbg("skipped %d class/vendor specific interface descriptors",
                                                                    numskipped);
1329
1330            /* Copy any unknown descriptors into a storage area for */
1331            /*  drivers to later parse */
1332            len = (int)(buffer - begin);
1333            if (!len) {
1334                ifp->extra = NULL;
1335                ifp->extralen = 0;
1336            } else {
1337                ifp->extra = kmalloc(len, GFP_KERNEL);
1338
```

```
1339                if (!ifp->extra) {
1340                    err("couldn't allocate memory for interface extra descriptors");
1341                    ifp->extralen = 0;
1342                    return -1;
1343                }
1344                memcpy(ifp->extra, begin, len);
1345                ifp->extralen = len;
1346            }
1347
```

在 usb_parse_configuration( )中，根据设备在具体配置下接口的个数，为 usb_interface 结构分配了空间，可是还没有为 usb_interface_descriptor 结构分配空间。一般的接口最多可以有 4 个接口描述块，这里的常数 USB_ALTSETTINGALLOC 定义为 4。代码中先按 4 个 usb_interface_descriptor 结构分配空间，然后通过一个 while 循环从前面读自目标设备的缓冲区中寻找和抽取有关的信息。当一个接口中有多于 4 个设置的时候，允许再增加 4 个(1271 行)，所以要重新分配一块连续的空间。这里的常数 SB_ALTSETTINGALLOC 也定义为 4，再多就不允许了。

在 while 循环中，首先从缓冲区中复制一个接口描述块，并相应地调整指针 buffer 和 parsed，以及剩余部分的大小。然后就来分析处理这个描述块所提供的信息。

每个接口描述块的开头是两个字节的 usb_descriptor_header 数据结构，说明了描述块的大小与类型，定义于 include/linux/usb.h：

```
212     /* All standard descriptors have these 2 fields in common */
213     struct usb_descriptor_header {
214             __u8  bLength;
215             __u8  bDescriptorType;
216     } __attribute__ ((packed));
```

如果描述块的类型为 USB_DT_INTERFACE、USB_DT_ENDPOINT、USB_DT_CONFIG、以及 USB_DT_DEVICE 这 4 者之一，就是一个 USB 标准的描述块，否则就把它跳过，继续往下看下一个描述块(1320～1324 行，以及 1304 行)。被跳过的描述块都是非 USB 标准的，一般是由具体厂商或行业（如数字照相机行业等等）自行定义的。对于这些非 USB 标准的描述块，一方面要通过 dbg( )予以报告；另一方面，更重要的是，要分配缓冲区将这些描述块保存下来(1332～1346 行)，以备将来由具体设备的驱动程序作进一步的分析，并使当前 usb_interface_descriptor 数据结构中的指针 extra 指向这块缓冲区。

至此，我们已经从原始缓冲区中找到了一个 USB 标准的描述块，下面就要根据这个描述块的类型来决定怎样继续往下分析了。我们继续看 usb_parse_interface( )的代码(drivers/usb/usb.c)。

[uhci_pci_probe( ) > setup_uhci( ) > uhci_start_root_hub( ) > usb_new_device( ) > usb_get_configuration( ) > usb_parse_configuration( ) > usb_parse_interface( )]

```
1348            /* Did we hit an unexpected descriptor? */
1349            header = (struct usb_descriptor_header *)buffer;
1350            if ((size >= sizeof(struct usb_descriptor_header)) &&
```

```
1351                ((header->bDescriptorType == USB_DT_CONFIG) ||
1352                 (header->bDescriptorType == USB_DT_DEVICE)))
1353                return parsed;
1354
1355        if (ifp->bNumEndpoints > USB_MAXENDPOINTS) {
1356                warn("too many endpoints");
1357                return -1;
1358        }
1359
1360        ifp->endpoint = (struct usb_endpoint_descriptor *)
1361                kmalloc(ifp->bNumEndpoints *
1362                sizeof(struct usb_endpoint_descriptor), GFP_KERNEL);
1363        if (!ifp->endpoint) {
1364                err("out of memory");
1365                return -1;
1366        }
1367
1368        memset(ifp->endpoint, 0, ifp->bNumEndpoints *
1369                sizeof(struct usb_endpoint_descriptor));
1370
1371        for (i = 0; i < ifp->bNumEndpoints; i++) {
1372                header = (struct usb_descriptor_header *)buffer;
1373
1374                if (header->bLength > size) {
1375                        err("ran out of descriptors parsing");
1376                        return -1;
1377                }
1378
1379                retval = usb_parse_endpoint(dev, ifp->endpoint + i, buffer, size);
1380                if (retval < 0)
1381                        return retval;
1382
1383                buffer += retval;
1384                parsed += retval;
1385                size -= retval;
1386        }
1387
1388        /* We check to see if it's an alternate to this one */
1389        ifp = (struct usb_interface_descriptor *)buffer;
1390        if (size < USB_DT_INTERFACE_SIZE ||
1391                ifp->bDescriptorType != USB_DT_INTERFACE ||
1392                !ifp->bAlternateSetting)
1393                return parsed;
1394    }
1395
1396    return parsed;
1397 }
```

如果原始缓冲区中下一个描述块的类型是 USB_DT_CONFIG 或 USB_DT_DEVICE，那就说明对一个新的配置或设备的描述开始了，当前配置中已经再没有更多的接口，所以便返回此时在原始缓冲区中的位置(1353 行)，供更高层的函数继续往下分析。否则，如果下一个描述块的类型是 USB_DT_INTERFACE 或 USB_DT_ENDPOINT，便找到了对下一个接口或端点的描述块。在每一个接口描述块的后面是若干端点描述块。

根据已经读入 usb_interface_descriptor 数据结构的信息，可以知道当前“设置”中有几个端点，从而为这些端点的 usb_endpoint_descriptor 数据结构分配空间(1360 行)。这种数据结构是对端点的抽象，定义于 include/linux/usb.h：

```
236    /* Endpoint descriptor */
237    struct usb_endpoint_descriptor {
238        __u8  bLength          __attribute__  ((packed));
239        __u8  bDescriptorType  __attribute__  ((packed));
240        __u8  bEndpointAddress __attribute__  ((packed));
241        __u8  bmAttributes     __attribute__  ((packed));
242        __u16 wMaxPacketSize   __attribute__  ((packed));
243        __u8  bInterval        __attribute__  ((packed));
244        __u8  bRefresh         __attribute__  ((packed));
245        __u8  bSynchAddress    __attribute__  ((packed));
246
247        unsigned char *extra;   /* Extra descriptors */
248        int extralen;
249    };
```

每一个端点都有个固定的端点号，在不同的配置中一个端点可以属于不同的接口，但是其端点号不会改变。另一方面，各个端点的最大信包容量、中断传输的应有周期等等参数也各不相同。

然后，就是通过一个 for 循环依次辩认和抽取各个端点描述块了。同样，每个端点描述块的开头也是由一个 usb_descriptor_header 数据结构说明了描述块的大小。对端点描述块的分析是由 usb_parse_endpoint( )完成的，其代码在 drivers/usb/usb.c 中：

[uhci_pci_probe( ) > setup_uhci( ) > uhci_start_root_hub( ) > usb_new_device( ) > usb_get_configuration( ) > usb_parse_configuration( ) > usb_parse_interface( ) > usb_parse_endpoint( )]

```
1161    static int usb_parse_endpoint(struct usb_device *dev,
            struct usb_endpoint_descriptor *endpoint, unsigned char *buffer, int size)
1162    {
1163        struct usb_descriptor_header *header;
1164        unsigned char *begin;
1165        int parsed = 0, len, numskipped;
1166
1167        header = (struct usb_descriptor_header *)buffer;
1168
1169        /* Everything should be fine being passed into here, but we sanity */
1170        /*  check JIC */
```

```
1171        if (header->bLength > size) {
1172            err("ran out of descriptors parsing");
1173            return -1;
1174        }
1175
1176        if (header->bDescriptorType != USB_DT_ENDPOINT) {
1177            warn(
            "unexpected descriptor 0x%X, expecting endpoint descriptor, type 0x%X",
1178                endpoint->bDescriptorType, USB_DT_ENDPOINT);
1179            return parsed;
1180        }
1181
1182        if (header->bLength == USB_DT_ENDPOINT_AUDIO_SIZE)
1183            memcpy(endpoint, buffer, USB_DT_ENDPOINT_AUDIO_SIZE);
1184        else
1185            memcpy(endpoint, buffer, USB_DT_ENDPOINT_SIZE);
1186
1187        le16_to_cpus(&endpoint->wMaxPacketSize);
1188
1189        buffer += header->bLength;
1190        size -= header->bLength;
1191        parsed += header->bLength;
1192
1193        /* Skip over the rest of the Class Specific or Vendor Specific */
1194        /*  descriptors */
1195        begin = buffer;
1196        numskipped = 0;
1197        while (size >= sizeof(struct usb_descriptor_header)) {
1198            header = (struct usb_descriptor_header *)buffer;
1199
1200            if (header->bLength < 2) {
1201                err("invalid descriptor length of %d", header->bLength);
1202                return -1;
1203            }
1204
1205            /* If we find another descriptor which is at or below us */
1206            /*  in the descriptor heirarchy then we're done  */
1207            if ((header->bDescriptorType == USB_DT_ENDPOINT) ||
1208                (header->bDescriptorType == USB_DT_INTERFACE) ||
1209                (header->bDescriptorType == USB_DT_CONFIG) ||
1210                (header->bDescriptorType == USB_DT_DEVICE))
1211                break;
1212
1213            dbg("skipping descriptor 0x%X",
1214                header->bDescriptorType);
1215            numskipped++;
1216
1217            buffer += header->bLength;
```

```
1218                size -= header->bLength;
1219                parsed += header->bLength;
1220            }
1221            if (numskipped)
1222                dbg("skipped %d class/vendor specific endpoint descriptors", numskipped);
1223
1224            /* Copy any unknown descriptors into a storage area for drivers */
1225            /*  to later parse */
1226            len = (int)(buffer - begin);
1227            if (!len) {
1228                endpoint->extra = NULL;
1229                endpoint->extralen = 0;
1230                return parsed;
1231            }
1232
1233            endpoint->extra = kmalloc(len, GFP_KERNEL);
1234
1235            if (!endpoint->extra) {
1236                err("couldn't allocate memory for endpoint extra descriptors");
1237                endpoint->extralen = 0;
1238                return parsed;
1239            }
1240
1241            memcpy(endpoint->extra, begin, len);
1242            endpoint->extralen = len;
1243
1244            return parsed;
1245    }
```

这里所期待的是端点描述块，所以如果描述块的类型不是 USB_DT_ENDPOINT，就立即返回，让高层的函数继续往下分析。一般端点描述块的大小是 7 个字节，但如果是用于音频设备的端点则有 9 个字节。把描述块开头的 7 个或 9 个字节复制到 usb_endpoint_descriptor 数据结构中，并相应推进 buffer 和 parsed，调整还剩下的 size 以后，对一个端点描述块的处理就结束了。但是，在端点描述块这一层上也会有非标准的、由厂家或有关行业自行定义的描述块，所以也要跳过这些描述块，并像前面看到的那样一方面通过 dbg( )提出报告，一方面将这些描述块复制下来，留待将来由具体的设备驱动程序加以分析。

回到 usb_parse_interface( )的代码中，在穷尽了一个(接口)设置的所有端点描述块以后，就完成了对这个设置的处理。如果原始缓冲区中还有足够的内容，下一个描述块又是个接口描述块，就可以回到 while 循环(1265 行)的开头，继续扫描原始缓冲区中剩下的内容，处理下一个设置了。

处理完一个配置的所有接口以后，便从 usb_parse_configuration( )返回到 usb_get_configuration( )中，继续读入和处理下一个配置，直到完成对设备的所有配置的处理。最后，返回到 usb_new_device( )的代码中(2135 行)，下一步是通过 usb_set_configuration( )将目标设备设置成采用 0 号配置，这是设备默认的配置。这个函数的代码在 drivers/usb/usb.c 中：

[uhci_pci_probe( )>setup_uhci( ) > uhci_start_root_hub( ) > usb_new_device( ) > usb_set_configuration( )]

```
1884    int usb_set_configuration(struct usb_device *dev, int configuration)
1885    {
1886        int i, ret;
1887        struct usb_config descriptor *cp = NULL;
1888
1889        for (i=0; i<dev->descriptor.bNumConfigurations; i++) {
1890            if (dev->config[i].bConfigurationValue == configuration) {
1891                cp = &dev->config[i];
1892                break;
1893            }
1894        }
1895        if (!cp) {
1896            warn("selecting invalid configuration %d", configuration);
1897            return -EINVAL;
1898        }
1899
1900        if ((ret = usb_control_msg(dev, usb_sndctrlpipe(dev, 0),
1901          USB_REQ_SET_CONFIGURATION, 0, configuration, 0, NULL, 0, HZ * SET_TIMEOUT)
                                                                                ) < 0)
1902            return ret;
1903
1904        dev->actconfig = cp;
1905        dev->toggle[0] = 0;
1906        dev->toggle[1] = 0;
1907        usb_set_maxpacket(dev);
1908
1909        return 0;
1910    }
```

至此，作为 PCI 设备，并且作为 USB 控制器的枚举与初始化已经完成了。但是，USB 主控制器必须与总线的根集中器配合才能进行有意义的操作。根集中器是 USB 主控制器的“大门”。而上述过程尚未触及对根集中器的初始化。事实上，根集中器和 USB 总线控制器在物理上总是集成在同一设备中，是同一设备的两个不同“接口”。另一方面，根集中器又是 USB 集中器的一种，在逻辑上和一般的 USB 集中器并无不同，一样适用 USB 集中器的驱动模块。所以对根集中器的初始化只有在安装了 USB 集中器的驱动模块以后才能进行。然而，对 USB 控制器的枚举与初始化跟 USB 集中器驱动模块的安装是两个独立的事件，并无保证何者在先，何者在后。所以，就有了两种可能的情况和对策。

如果 USB 控制器的枚举与初始化在先，则只好推迟 USB 集中器的初始化。到安装 USB 集中器的驱动模块的时候，可以让它来“认领”已经枚举的根集中器，然后对其进行初始化。对于根集中器，这是可能性最大的情况。

如果 USB 集中器的驱动模块安装在先，则安装时认领不到 USB 集中器，因而无事可做。然后，在完成了 USB 总线控制器的枚举与初始化，并检测到一个具体 USB 集中器的存在时，就设法找到其驱动模块，再通过该驱动模块完成其初始化。根集中器是特殊的 USB 集中器，它的存在是不需要检测

的，有 USB 总线控制器就必有根集中器，但是一般的集中器就不同了。实际上，一般 USB 设备的枚举与具体驱动模块的安装更加随机，因为 USB 总线允许“热插入”，随时可以把设备插上拔下。

所以，现在就通过 usb_find_drivers( )来找找，看 USB 集中器的驱动模块是否已经安装。这个函数的代码在 drivers/usb/usb.c 中：

[uhci_pci_probe( ) > setup_uhci( ) > uhci_start_root_hub( ) > usb_new_device( ) > usb_find_drivers( )]

```
833    /*
834     * This entrypoint gets called for each new device.
835     *
836     * All interfaces are scanned for matching drivers.
837     */
838    static void usb_find_drivers(struct usb_device *dev)
839    {
840        unsigned ifnum;
841        unsigned rejected = 0;
842        unsigned claimed = 0;
843
844        for (ifnum = 0; ifnum < dev->actconfig->bNumInterfaces; ifnum++) {
845            /* if this interface hasn't already been claimed */
846            if (!usb_interface_claimed(dev->actconfig->interface + ifnum)) {
847                if (usb_find_interface_driver(dev, ifnum))
848                    rejected++;
849                else
850                    claimed++;
851            }
852        }
853
854        if (rejected)
855            dbg("unhandled interfaces on device");
856
857        if (!claimed) {
858            warn(
        "USB device %d (vend/prod 0x%x/0x%x) is not claimed by any active driver.",
859                dev->devnum,
860                dev->descriptor.idVendor,
861                dev->descriptor.idProduct);
862    #ifdef DEBUG
863            usb_show_device(dev);
864    #endif
865        }
866    }
```

对于 USB 主控制器设备的每一个接口（其中之一必定是根集中器），只要尚未被认领，就通过 usb_find_interface_driver( )扫描已经安装的驱动模块队列并进行比对。其代码在同一文件中：

[uhci_pci_probe( ) > setup_uhci( ) > uhci_start_root_hub( ) > usb_new_device( ) > usb_find_drivers( ) > usb_find_interface_driver( )]

```
623     /*
624      * This entrypoint gets called for each new device.
625      *
626      * We now walk the list of registered USB drivers,
627      * looking for one that will accept this interface.
628      *
629      * "New Style" drivers use a table describing the devices and interfaces
630      * they handle.  Those tables are available to user mode tools deciding
631      * whether to load driver modules for a new device.
632      *
633      * The probe return value is changed to be a private pointer.  This way
634      * the drivers don't have to dig around in our structures to set the
635      * private pointer if they only need one interface.
636      *
637      * Returns: 0 if a driver accepted the interface, -1 otherwise
638      */
639     static int usb_find_interface_driver(struct usb_device *dev, unsigned ifnum)
640     {
641         struct list_head *tmp;
642         struct usb_interface *interface;
643         void *private;
644         const struct usb_device_id *id;
645         struct usb_driver *driver;
646         int i;
647
648         if ((!dev) || (ifnum >= dev->actconfig->bNumInterfaces)) {
649             err("bad find_interface_driver params");
650             return -1;
651         }
652
653         interface = dev->actconfig->interface + ifnum;
654
655         if (usb_interface_claimed(interface))
656             return -1;
657
658         private = NULL;
659         for (tmp = usb_driver_list.next; tmp != &usb_driver_list;) {
660
661             driver = list_entry(tmp, struct usb_driver, driver_list);
662             tmp = tmp->next;
663
664             down(&driver->serialize);
665             id = driver->id_table;
666             /* new style driver? */
667             if (id) {
```

```
668                 for (i = 0; i < interface->num_altsetting; i++) {
669                     interface->act altsetting = i;
670                     id = usb match_id(dev, interface, id);
671                     if (id) {
672                         private = driver->probe(dev, ifnum, id);
673                         if (private != NULL)
674                             break;
675                     }
676                 }
677                 /* if driver not bound, leave defaults unchanged */
678                 if (private == NULL)
679                     interface->act_altsetting = 0;
680             }
681             else /* "old style" driver */
682                 private = driver->probe(dev, ifnum, NULL);
683 
684             up(&driver->serialize);
685             if (private) {
686                 usb_driver_claim_interface(driver, interface, private);
687                 return 0;
688             }
689         }
690 
691         return -1;
692     }
```

每个已安装的驱动模块都有个“比对表”，说明本模块适用于哪一些或什么样的设备，可以通过usb_match_id( )与具体设备提供的数据比对。我们将在后面结合扫描器的驱动列出这个函数的代码。比对成功，即找到了适用的驱动模块以后，就对目标设备（接口）执行由驱动模块通过函数指针 probe 提供的操作，完成目标设备（接口）的初始化。

在我们这个情景中，假定 USB 集中器的驱动模块尚未安装，所以只好暂缓，等待驱动模块的安装或 USB 集中器初始化的进行。不过，usb_new_device( )的代码中还调用了一个函数 call_policy( )，这涉及专为“热插入”而设计的一种机制。其大致的作用和过程是让 call_policy( )构筑一个命令行“/sbin/hotplug usb”以及必要的环境变量，然后创建起一个内核线程，再让这个线程升级成为一个进程，并执行工具软件/sbin/hotplug 以装入 USB 集中器的驱动模块。限于篇幅，我们这里就从略了。

不管是否出于 call_policy( )的间接启动，我们假定 USB 总线的驱动模块或迟或早都会得到安装。安装 USB 总线的驱动模块时，就会执行其初始化程序 usb_init( )。其代码在 drivers/usb/usb.c 中：

```
2243    static int __init usb_init(void)
2244    {
2245        usb_major_init( );
2246        usbdevfs_init( );
2247        usb_hub_init( );
2248 
```

```
2249        return 0;
2250    }
```

就像别的设备驱动程序一样，首先要向系统登记，这是由 usb_major_init( )完成的。其代码在 drivers/usb/usb.c 中：

[usb_init( ) > usb_major_init( )]

```
2208    int usb_major_init(void)
2209    {
2210        if (devfs_register_chrdev(USB_MAJOR, "usb", &usb_fops)) {
2211            err("unable to get major %d for usb devices", USB_MAJOR);
2212            return -EBUSY;
2213        }
2214
2215        usb_devfs_handle = devfs_mk_dir(NULL, "usb", NULL);
2216
2217        return 0;
2218    }
```

对这段代码已经没有什么可说的了。此外，上面 2246 行 usbdevfs_init( )的作用只是向 devfs 特殊文件系统登记，读者可参阅"设备文件系统 devfs"一节。所以，usb_init( )中关键的操作就是 usb_hub_init( )。这个函数的代码在 drivers/usb/hub.c 中：

[usb_init( ) > usb_hub_init( )]

```
786     int usb_hub_init(void)
787     {
788         int pid;
789
790         if (usb_register(&hub_driver) < 0) {
791             err("Unable to register USB hub driver");
792             return -1;
793         }
794
795         pid = kernel_thread(usb_hub_thread, NULL,
796             CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
797         if (pid >= 0) {
798             khubd_pid = pid;
799
800             return 0;
801         }
802
803         /* Fall through if kernel thread failed */
804         usb_deregister(&hub_driver);
805         err("failed to start usb hub thread");
```

```
806
807         return -1;
808     }
```

首先通过 usb_register( )登记一个 USB 设备驱动模块。每一种 USB 设备都有一个 usb_driver 数据结构，USB 集中器的 usb_driver 数据结构是 hub_driver，定义于 drivers/usb/hub.c：

```
775     static struct usb_driver hub_driver = {
776         name:       "hub",
777         probe:      hub_probe,
778         ioctl:      hub_ioctl,
779         disconnect: hub_disconnect,
780         id_table:   hub_id_table,
781     };
```

我们将在后面以扫描器为实例讲述 USB 设备的初始化时，详细介绍 usb_register( )，这里先简略地说明一下。所有的 USB 设备都采用相同的主设备号 USB_MAJOR，而不同类的 USB 设备则由次设备号区分。内核中有个以次设备号为下标的 usb_driver 结构指针数组 usb_minors[ ]，登记时就根据具体 usb_driver 结构中提供的次设备号将其起始地址填入这个数组中。上面 hub_driver 的定义中没有列出其次设备号，实际上表示 USB 集中器的次设备号为 0。此外，内核中还有个 USB 驱动模块队列 usb_driver_list，登记时把 usb_driver 结构通过其队列头 driver_list 挂入这个队列。然后，还要对已经枚举的所有 USB 设备的 usb_device 结构进行一趟扫描，让新登记的驱动模块“认领”应该由它驱动的设备（或接口），如果找到了就对其执行一次由驱动模块提供的 probe 操作。在我们现在这个情景中，已经枚举的 USB 设备只有一个，那就是根集中器，所以对其调用由 hub_driver 提供的 probe 函数 hub_probe( )。其代码在 drivers/usb/hub.c 中（下面的调用路径中跳过了几个函数，后面还会介绍）。

[usb_init( ) > usb_hub_init( ) > usb_register( ) > usb_scan_devices( ) > usb_check_support( )
 > usb_find_interface_driver( ) > hub_probe( )]

```
238     static void *hub_probe(struct usb_device *dev, unsigned int i,
239                     const struct usb_device_id *id)
240
241     {
242         struct usb_interface_descriptor *interface;
243         struct usb_endpoint descriptor *endpoint;
244         struct usb_hub *hub;
245         unsigned long flags;
246
247         interface = &dev->actconfig->interface[i].altsetting[0];
248
249         /* Some hubs have a subclass of 1, which AFAICT according to the */
250         /*  specs is not defined, but it works */
251         if ((interface->bInterfaceSubClass != 0) &&
252             (interface->bInterfaceSubClass != 1)) {
253             err("invalid subclass (%d) for USB hub device #%d",
```

```
254                interface->bInterfaceSubClass, dev->devnum);
255            return NULL;
256        }
257
258        /* Multiple endpoints? What kind of mutant ninja-hub is this? */
259        if (interface->bNumEndpoints != 1) {
260            err("invalid bNumEndpoints (%d) for USB hub device #%d",
261                interface->bNumEndpoints, dev->devnum);
262            return NULL;
263        }
264
265        endpoint = &interface->endpoint[0];
266
267        /* Output endpoint? Curiousier and curiousier.. */
268        if (!(endpoint->bEndpointAddress & USB_DIR_IN)) {
269            err("Device #%d is hub class, but has output endpoint?",
270                dev->devnum);
271            return NULL;
272        }
273
274        /* If it's not an interrupt endpoint, we'd better punt! */
275        if ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) !=
                                            USB_ENDPOINT_XFER_INT) {
276            err("Device #%d is hub class, but has endpoint other than interrupt?",
277                dev->devnum);
278            return NULL;
279        }
280
281        /* We found a hub */
282        info("USB hub found");
283
284        hub = kmalloc(sizeof(*hub), GFP_KERNEL);
285        if (!hub) {
286            err("couldn't kmalloc hub struct");
287            return NULL;
288        }
289
290        memset(hub, 0, sizeof(*hub));
291
292        INIT_LIST_HEAD(&hub->event_list);
293        hub->dev = dev;
294
295        /* Record the new hub's existence */
296        spin_lock_irqsave(&hub_event_lock, flags);
297        INIT_LIST_HEAD(&hub->hub_list);
298        list_add(&hub->hub_list, &hub_list);
299        spin_unlock_irqrestore(&hub_event_lock, flags);
300
```

```
301         if (usb_hub_configure(hub, endpoint) >= 0)
302             return hub;
303
304         err("hub configuration failed for device #%d", dev->devnum);
305
306         /* free hub, but first clean up its list. */
307         spin_lock_irqsave(&hub_event_lock, flags);
308
309         /* Delete it and then reset it */
310         list_del(&hub->event_list);
311         INIT_LIST_HEAD(&hub->event_list);
312         list_del(&hub->hub_list);
313         INIT_LIST_HEAD(&hub->hub_list);
314
315         spin_unlock_irqrestore(&hub_event_lock, flags);
316
317         kfree(hub);
318
319         return NULL;
320     }
```

参数 dev 指向具体设备(在这里是根集中器)的 usb_device 数据结构，i 表明当前处理的接口号。当设备中有多个接口时，对每个接口都会调用一次相应的 probe 函数。参数 id 则指向一个用于比对、认领的 usb_device_id 数据结构，里面有设备驱动模块所适用的设备类型、制造厂商、产品编号等等信息。由于具体 probe 函数的执行实际上是针对接口(即逻辑设备)的，所以先根据接口号从设备的当前配置中取得指向具体接口描述信息的指针 interface。在调用这个函数之前，已经对设备进行了比对，根据其设备类型、制造厂商等等信息初步认定是个 USB 集中器，这才调用 hub_probe( )的。但是，这里还要再进一步作一些检验。USB 集中器的子设备类型号应该是 0 或 1，除默认的控制端点以外应该只有一个中断交互输入端点，251～279 行对这些特征加以检验。如果通过了所有这些检验，就最后认定了这是个 USB 集中器。虽然 usb_device 数据结构中包含了目标设备作为一般 USB 设备所共有的各种信息，作为具体的 USB 集中器设备还有附加的、反映此种设备特性的信息，以及运行所需的一些字段或成分，所以还要为其分配一个 usb_hub 数据结构。这种数据结构定义于 drivers/usb/hub.h:

```
 93     struct usb_hub {
 94         struct usb_device *dev;
 95
 96         struct urb *urb;        /* Interrupt polling pipe */
 97
 98         char buffer[(USB_MAXCHILDREN + 1 + 7) / 8];
                                /* add 1 bit for hub status change */
 99                             /* and add 7 bits to round up to byte boundary */
100         int error;
101         int nerrors;
102
103         struct list_head hub_list;
```

```
104
105         struct list_head event_list;
106
107         /* Number of ports on the hub */
108         int nports;
109
110         struct usb_hub_descriptor *descriptor;
111     };
```

结构中有个位图 buffer[ ]，除其中一位用作总的状态改变标志位以外，每一位都对应着 USB 集中器上的一个端口，反映着该端口的状态，字段 nports 则说明端口的数量。指针 descriptor 指向一个 usb_hub_descriptor 数据结构，定义于同一文件中：

```
77      /* Hub descriptor */
78      struct usb_hub_descriptor {
79          __u8  bLength;
80          __u8  bDescriptorType;
81          __u8  bNbrPorts;
82          __u16 wHubCharacteristics;
83          __u8  bPwrOn2PwrGood;
84          __u8  bHubContrCurrent;
85
86          /* DeviceRemovable and PortPwrCtrlMask want to be variable-length
87             bitmaps that hold max 256 entries, but for now they're ignored */
88          __u8  bitmap[0];
89      } __attribute__ ((packed));
```

显然，这些信息只能来自设备本身。此外，对 USB 集中器的初始化还应该包括为通过周期性的中断传输查询其状态变化作出安排。所以，在对 usb_hub 结构进行一些初步的初始化以后，便通过 usb_hub_configure( )进一步完成其初始化。其代码在 drivers/usb/hub.c 中：

[usb_init( ) > usb_hub_init( ) > usb_register( ) > usb_scan_devices( ) > usb_check_support( ) > usb_find_interface_driver( ) > hub_probe( ) > usb_hub_configure( )]

```
125     static int usb_hub_configure(struct usb_hub *hub,
                                    struct usb_endpoint_descriptor *endpoint)
126     {
127         struct usb_device *dev = hub->dev;
128         struct usb_hub_status hubstatus;
129         char portstr[USB_MAXCHILDREN + 1];
130         unsigned int pipe;
131         int i, maxp, ret;
132
133         hub->descriptor = kmalloc(HUB_DESCRIPTOR_MAX_SIZE, GFP_KERNEL);
134         if (!hub->descriptor) {
135             err("Unable to kmalloc %d bytes for hub descriptor",
```

```
                                                        HUB_DESCRIPTOR_MAX_SIZE);
136             return -1;
137         }
138 
139         /* Request the entire hub descriptor. */
140         ret = usb_get_hub descriptor(dev, hub->descriptor, HUB_DESCRIPTOR_MAX_SIZE);
141             /* <hub->descriptor> is large enough for a hub with 127 ports;
142              * the hub can/will return fewer bytes here. */
143         if (ret < 0) {
144             err("Unable to get hub descriptor (err = %d)", ret);
145             kfree(hub->descriptor);
146             return -1;
147         }
148 
149         hub->nports = dev->maxchild = hub->descriptor->bNbrPorts;
150         info("%d port%s detected", hub->nports, (hub->nports == 1) ? "" : "s");
151 
152         if (hub->descriptor->wHubCharacteristics & HUB_CHAR_COMPOUND)
153             dbg("part of a compound device");
154         else
155             dbg("standalone hub");
156 
157         switch (hub->descriptor->wHubCharacteristics & HUB_CHAR_LPSM) {
158             case 0x00:
159                 dbg("ganged power switching");
160                 break;
161             case 0x01:
162                 dbg("individual port power switching");
163                 break;
164             case 0x02:
165             case 0x03:
166                 dbg("unknown reserved power switching mode");
167                 break;
168         }
169 
170         switch (hub->descriptor->wHubCharacteristics & HUB_CHAR_OCPM) {
171             case 0x00:
172                 dbg("global over-current protection");
173                 break;
174             case 0x08:
175                 dbg("individual port over-current protection");
176                 break;
177             case 0x10:
178             case 0x18:
179                 dbg("no over-current protection");
180                         break;
181         }
182 
```

```
183         dbg("power on to power good time: %dms", hub->descriptor->bPwrOn2PwrGood * 2);
184         dbg("hub controller current requirement: %dmA",
                                             hub->descriptor->bHubContrCurrent);
185 
186         for (i = 0; i < dev->maxchild; i++)
187             portstr[i] =
                    hub->descriptor->bitmap[((i + 1) / 8)] & (1 << ((i + 1) % 8)) ?
                                                                    'F' : 'R';
188         portstr[dev->maxchild] = 0;
189 
190         dbg("port removable status: %s", portstr);
191 
192         ret = usb_get_hub_status(dev, &hubstatus);
193         if (ret < 0) {
194             err("Unable to get hub status (err = %d)", ret);
195             kfree(hub->descriptor);
196             return -1;
197         }
198 
199         le16 to_cpus(&hubstatus.wHubStatus);
200 
201         dbg("local power source is %s",
202             (hubstatus.wHubStatus & HUB_STATUS_LOCAL_POWER) ?
                                             "lost (inactive)" : "good");
203 
204         dbg("%sover-current condition exists",
205             (hubstatus.wHubStatus & HUB_STATUS_OVERCURRENT) ? "" : "no ");
206 
207         /* Start the interrupt endpoint */
208         pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
209         maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
210 
211         if (maxp > sizeof(hub->buffer))
212             maxp = sizeof(hub->buffer);
213 
214         hub->urb = usb_alloc_urb(0);
215         if (!hub->urb) {
216             err("couldn't allocate interrupt urb");
217             kfree(hub->descriptor);
218             return -1;
219         }
220 
221         FILL_INT_URB(hub->urb, dev, pipe, hub->buffer, maxp, hub_irq,
222             hub, endpoint->bInterval);
223         ret = usb_submit_urb(hub->urb);
224         if (ret) {
225             err("usb_submit_urb failed (%d)", ret);
226             kfree(hub->descriptor);
```

```
227            return -1;
228        }
229
230        /* Wake up khubd */
231        wake up(&khubd wait);
232
233        usb hub power on(hub);
234
235        return 0;
236    }
```

首先为 usb_hub_descriptor 数据结构分配空间，再通过 usb_get_hub_descriptor( )从集中器读入所需的信息，即集中器描述块。这个函数的代码在 drivers/usb/hub.c 中：

[usb_init( ) > usb_hub_init( ) > usb_register( ) > usb_scan_devices( ) > usb_check_support( ) > usb_find_interface_driver( ) > hub_probe( ) > usb_hub_configure( ) > usb_get_hub_descriptor( )]

```
41    static int usb_get_hub_descriptor(struct usb device *dev, void *data, int size)
42    {
43        return usb control msg(dev, usb rcvctrlpipe(dev, 0),
44            USB REQ GET DESCRIPTOR, USB DIR IN | USB RT HUB,
45            USB DT HUB << 8, 0, data, size, HZ);
46    }
```

可见，所需的信息是通过一次控制传输从集中器读入的。读入了集中器描述块以后。就知道了这个集中器有几个端口，也知道了它的其它一些特性。例如，一个集中器可以是一个单纯的 USB 集中器，也可以是一个复合设备中的一部分；集中器可以是自带电源的，也可以通过 USB 电缆从主机吸取电流；端口上可以带有过电流保护，也可以不带。不过我们在这里对这些特性不感兴趣。

接着，再通过 usb_get_hub_status( )启动一次控制传输，进一步从集中器读入状态信息(drivers/usb/hub.c)，以获取有关其电源供应的当前状况。

[usb_init( ) > usb_hub_init( ) > usb_register( ) > usb_scan_devices( ) > usb_check_support( ) > usb_find_interface_driver( ) > hub_probe( ) > usb_hub_configure( ) > usb_get_hub_status( )]

```
66    static int usb_get_hub_status(struct usb device *dev, void *data)
67    {
68        return usb control msg(dev, usb rcvctrlpipe(dev, 0),
69            USB_REQ_GET_STATUS, USB_DIR_IN | USB_RT_HUB, 0, 0,
70            data, sizeof(struct usb hub status), HZ);
71    }
```

当然，我们对电源也不感兴趣。

下面是对中断交互的安排，这才是关键性的。对于 USB 集中器，要为其安排一个或者说“调度”一个周期性的中断传输(每个中断传输中只有一个交互)，使 USB 总线控制器能周期地查询其状态，让它有机会报告各个端口的状态变化。

我们将在后面结合扫描器的驱动介绍对中断传输的调度，这里先作一些简短的说明。

首先要通过 usb_alloc_urb( )分配一个"USB 传输请求块"，即 usb 数据结构。再通过宏操作 FILL_INT_URB( )设置好这个数据结构，设置的内容包括对方的设备地址与端点号，用来接收信息的缓冲区 hub->buffer，指向具体 usb_hub 数据结构的指针 hub，以及当接收到来自集中器的信息以后需要执行的"中断服务程序"hub_irq( )、查询的周期等等。最后通过 usb_submit_urb( )提交这个请求，就是根据 urb 数据结构的内容创建一个交互请求，即 uhci_td 数据结构，并根据查询周期将其链入 skeltd[ ]中某两个元素之间。这样，USB 总线控制器便会周期性地执行这个交互请求，对目标集中器通过中断交互进行查询。

至此，通过 usb_register( )对根集中器的枚举与初始化已基本完成，我们随着 CPU 回到 usb_hub_init( )的代码中。

阅读过本章"PCI 总线"一节的读者可能已经注意到了，PCI 总线和 USB 总线的枚举过程有明显的不同。PCI 总线的枚举过程是对总线上所有设备的枚举，并且是个递归的过程。而对于 USB 总线，则至今只看到了根集中器的"枚举"，却未见有进一步对连接在集中器上的设备的枚举。虽然 USB 总线的结构也有递归性(多级集中器)，但却不见有递归的操作。这实际上反映了二者的一个重要的区别。对于 PCI 总线，虽然在一些特殊的系统中对一些特殊的部件(需要特殊的硬件结构)也有"热插入"的要求，但基本的假设是 PCI 设备在系统加电之前就已经静态地连接在总线上。而对 USB 总线却恰好相反，基本的假设是"热插入"，即多数 USB 设备都会在系统加电以后动态地加入或离开系统。在最为一般的情况下，系统初始化时 USB 总线上除根集中器外没有任何设备，而事先有设备连在 USB 总线上反倒是特例。

所以，对于根集中器"后面"的设备的枚举，是一项需要"细水长流"的任务。USB 集中器的驱动模块为此创建了一个内核线程 khubd，专门处理集中器的状态变化。每当将一个 USB 设备插入集中器时，集中器在下一次受到 USB 主控制器(通过中断交互)的查询时就会报告状态有了改变，从而 USB 主控制器在当前框架结束时会向 CPU 发出一个中断请求。而 USB 总线的中断服务程序，则会根据在该框架内完成的交互请求，从而 urb 数据结构的内容、调用具体设备的中断服务程序（以后读者会见到）。对于集中器的中断交互，这个中断服务程序是 hub_irq( )。其代码在 drivers/usb/hub.c 中：

```
80	static void hub_irq(struct urb *urb)
81	{
82		struct usb_hub *hub = (struct usb_hub *)urb->context;
83		unsigned long flags;
84	
85		/* Cause a hub reset after 10 consecutive errors */
86		if (urb->status) {
87			if (urb->status == -ENOENT)
88				return;
89	
90			dbg("nonzero status in irq %d", urb->status);
91	
92			if ((++hub->nerrors < 10) || hub->error)
93				return;
94	
```

```
95              hub->error = urb->status;
96          }
97
98          hub->nerrors = 0;
99
100         /* Something happened, let khubd figure it out */
101         if (waitqueue_active(&khubd_wait)) {
102             /* Add the hub to the event queue */
103             spin_lock_irqsave(&hub_event_lock, flags);
104             if (list_empty(&hub->event_list)) {
105                 list_add(&hub->event_list, &hub_event_list);
106                 wake_up(&khubd_wait);
107             }
108             spin_unlock_irqrestore(&hub_event_lock, flags);
109         }
110     }
```

根据 urb 数据结构的内容可以找到目标集中器的 usb_hub 结构。如果 urb->status 为 0，就表示中断交互正常完成，并且集中器的状态有了变化(如果集中器的状态无变化，则集中器返回 NAK，此时 USB 控制器不会将该中交互描述块中的 TD_CTRL_ACTIVE 位清 0，也不会向 CPU 发出中断请求，见后)。只要 khubd 已经在运行，就把目标集中器的 usb_hub 结构通过其队列头 event_list 挂入 khubd 的等待队列 hub_event_list，然后唤醒 khubd。

线程 khubd 是在 usb_hub_init( )中(795 行)通过 kernel_thread( )创建的，其执行代码 usb_hub_thread( ) 在 drivers/usb/hub.c 中：

```
742     static int usb_hub_thread(void *__hub)
743     {
744         lock_kernel( );
745
746         /*
747          * This thread doesn't need any user-level access,
748          * so get rid of all our resources
749          */
750
751         daemonize( );
752
753         /* Setup a nice name */
754         strcpy(current->comm, "khubd");
755
756         /* Send me a signal to get me die (for debugging) */
757         do {
758             usb_hub_events( );
759             interruptible_sleep_on(&khubd_wait);
760         } while (!signal_pending(current));
761
762         dbg("usb_hub_thread exiting");
```

```
763
764         up_and_exit(&khubd_exited, 0);
765     }
```

代码中的 do-while 循环实际上是个无限循环，只要不向这个线程发送信号，就会一直循环下去。在循环体中，先通过 usb_hub_events( )检查和处理各个集中器(现在还只有根集中器)的状态变化，然后就睡眠（建议读者思考一下，在循环中为什么处理在先而睡眠在后），直到被唤醒而再次执行 usb_hub_events( )。这个函数的代码在 drivers/usb/hub.c 中：

[usb_hub_thread( ) > usb_hub_events( )]

```
622     static void usb_hub_events(void)
623     {
624         unsigned long flags;
625         struct list_head *tmp;
626         struct usb_device *dev;
627         struct usb_hub *hub;
628         struct usb_hub_status hubsts;
629         unsigned short hubstatus, hubchange;
630         int i, ret;
631
632         /*
633          *  We restart the list everytime to avoid a deadlock with
634          * deleting hubs downstream from this one. This should be
635          * safe since we delete the hub from the event list.
636          * Not the most efficient, but avoids deadlocks.
637          */
638         while (1) {
639             spin_lock_irqsave(&hub_event_lock, flags);
640
641             if (list_empty(&hub_event_list))
642                 goto he_unlock;
643
644             /* Grab the next entry from the beginning of the list */
645             tmp = hub_event_list.next;
646
647             hub = list_entry(tmp, struct usb_hub, event_list);
648             dev = hub->dev;
649
650             list_del(tmp);
651             INIT_LIST_HEAD(tmp);
652
653             spin_unlock_irqrestore(&hub_event_lock, flags);
654
655             if (hub->error) {
656                 dbg("resetting hub %d for error %d", dev->devnum, hub->error);
657
```

```
658                 if (usb_hub_reset(hub)) {
659                     err("error resetting hub %d - disconnecting", dev->devnum);
660                     usb_hub_disconnect(dev);
661                     continue;
662                 }
663 
664                 hub->nerrors = 0;
665                 hub->error = 0;
666             }
667 
668             for (i = 0; i < hub->nports; i++) {
669                 struct usb_port_status portsts;
670                 unsigned short portstatus, portchange;
671 
672                 ret = usb_get_port_status(dev, i + 1, &portsts);
673                 if (ret < 0) {
674                     err("get_port_status failed (err = %d)", ret);
675                     continue;
676                 }
677 
678                 portstatus = le16_to_cpu(portsts.wPortStatus);
679                 portchange = le16_to_cpu(portsts.wPortChange);
680 
681                 if (portchange & USB_PORT_STAT_C_CONNECTION) {
682                     dbg("port %d connection change", i + 1);
683 
684                     usb_hub_port_connect_change(dev, i, &portsts);
685                 } else if (portchange & USB_PORT_STAT_C_ENABLE) {
686                     dbg("port %d enable change, status %x", i + 1, portstatus);
687                     usb_clear_port_feature(dev, i + 1, USB_PORT_FEAT_C_ENABLE);
688 
689                     /*
690                      * EM interference sometimes causes bad shielded USB devices to
691                      * be shutdown by the hub, this hack enables them again.
692                      * Works at least with mouse driver.
693                      */
694                     if (!(portstatus & USB_PORT_STAT_ENABLE) &&
695                         (portstatus & USB_PORT_STAT_CONNECTION) &&
                                                        (dev->children[i])) {
696                         err(
            "already running port %i disabled by hub (EMI?), re-enabling...",
697                             i + 1);
698                         usb_hub_port_connect_change(dev, i, &portsts);
699                     }
700                 }
701 
702                 if (portchange & USB_PORT_STAT_C_SUSPEND) {
703                     dbg("port %d suspend change", i + 1);
```

```
704                 usb_clear_port_feature(dev, i + 1,  USB_PORT_FEAT_C_SUSPEND);
705             }
706
707             if (portchange & USB_PORT_STAT_C_OVERCURRENT) {
708                 err("port %d over-current change", i + 1);
709                 usb_clear_port_feature(dev, i + 1,
                                                  USB_PORT_FEAT_C_OVER_CURRENT);
710                 usb_hub_power_on(hub);
711             }
712
713             if (portchange & USB_PORT_STAT_C_RESET) {
714                 dbg("port %d reset change", i + 1);
715                 usb_clear_port_feature(dev, i + 1, USB_PORT_FEAT_C_RESET);
716             }
717         } /* end for i */
718
719         /* deal with hub status changes */
720         if (usb_get_hub_status(dev, &hubsts) < 0)
721             err("get_hub_status failed");
722         else {
723             hubstatus = le16_to_cpup(&hubsts.wHubStatus);
724             hubchange = le16_to_cpup(&hubsts.wHubChange);
725             if (hubchange & HUB_CHANGE_LOCAL_POWER) {
726                 dbg("hub power change");
727                 usb_clear_hub_feature(dev, C_HUB_LOCAL_POWER);
728             }
729             if (hubchange & HUB_CHANGE_OVERCURRENT) {
730                 dbg("hub overcurrent change");
731                 wait_ms(500);   /* Cool down */
732                 usb_clear_hub_feature(dev, C_HUB_OVER_CURRENT);
733                             usb_hub_power_on(hub);
734             }
735         }
736         } /* end while (1) */
737
738     he_unlock:
739         spin_unlock_irqrestore(&hub_event_lock, flags);
740     }
```

通过一个 while 循环，khubd 依次摘下并处理挂在 hub_event_list 中的每一个 usb_hub 数据结构。如果 hub->error 非 0，就表示已经连续出错 10 次以上，所以通过 usb_hub_reset( )向相应的集中器发出一个 reset 命令，让它“重新做人”。如果连这也失败，那就要通过 usb_hub_disconnect( )断开其连接。

然后，通过一个 for 循环检查和处理集中器的每个端口，因为至此只知道集中器的状态发生了变化，但并不知道具体的情况，那要通过控制传输进一步查询。所以，对于集中器的每个端口，通过 usb_get_port_status( )启动一次控制交互读入其状态信息。其代码见 drivers/usb/hub.c:

[usb_hub_thread( ) > usb_hub_events( ) > usb_get_port_status( )]

```
73      static int usb_get_port_status(struct usb_device *dev, int port, void *data)
74      {
75          return usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
76              USB_REQ_GET_STATUS, USB_DIR_IN | USB_RT_PORT, 0, port,
77              data, sizeof(struct usb_hub_status), HZ);
78      }
```

如果具体端口的状态信息表明其设备连接状态发生了变化，那就要作出进一步的处理和反应，这是通过 usb_hub_port_connect_change( )完成的。这个函数的代码在 drivers/usb/hub.c 中：

[usb_hub_thread( ) > usb_hub_events( ) > usb_hub_port_connect_change( )]

```
519     static void usb_hub_port_connect_change(struct usb_device *hub, int port,
520                         struct usb_port_status *portsts)
521     {
522         struct usb_device *dev;
523         unsigned short portstatus, portchange;
524         unsigned int delay = HUB_SHORT_RESET_TIME;
525         int i;
526         char *portstr, *tempstr;
527
528         portstatus = le16_to_cpu(portsts->wPortStatus);
529         portchange = le16_to_cpu(portsts->wPortChange);
530         dbg("port %d, portstatus %x, change %x, %s", port + 1, portstatus,
531             portchange, portstatus & (1 << USB_PORT_FEAT_LOWSPEED) ?
                                                        "1.5 Mb/s" : "12 Mb/s");
532
533         /* Clear the connection change status */
534         usb_clear_port_feature(hub, port + 1, USB_PORT_FEAT_C_CONNECTION);
535
536         /* Disconnect any existing devices under this port */
537         if (hub->children[port])
538             usb_disconnect(&hub->children[port]);
539
540         /* Return now if nothing is connected */
541         if (!(portstatus & USB_PORT_STAT_CONNECTION)) {
542             if (portstatus & USB_PORT_STAT_ENABLE)
543                 usb_hub_port_disable(hub, port);
544
545             return;
546         }
547
548         down(&usb_address0_sem);
549
550         tempstr = kmalloc(1024, GFP_KERNEL);
```

```
551         portstr = kmalloc(1024, GFP_KERNEL);
552 
553         for (i = 0; i < HUB_PROBE_TRIES; i++) {
554             struct usb_device *pdev, *cdev;
555 
556             /* Allocate a new device struct */
557             dev = usb_alloc_dev(hub, hub->bus);
558             if (!dev) {
559                 err("couldn't allocate usb_device");
560                 break;
561             }
562 
563             hub->children[port] = dev;
564 
565             /* Reset the device */
566             if (usb_hub_port_reset(hub, port, dev, delay)) {
567                 usb_free_dev(dev);
568                 break;
569             }
570 
571             /* Find a new device ID for it */
572             usb_connect(dev);
573 
574             /* Create a readable topology string */
575             cdev = dev;
576             pdev = dev->parent;
577             if (portstr && tempstr) {
578                 portstr[0] = 0;
579                 while (pdev) {
580                     int port;
581 
582                     for (port = 0; port < pdev->maxchild; port++)
583                         if (pdev->children[port] == cdev)
584                             break;
585 
586                     strcpy(tempstr, portstr);
587                     if (!strlen(tempstr))
588                         sprintf(portstr, "%d", port + 1);
589                     else
590                         sprintf(portstr, "%d/%s", port + 1, tempstr);
591 
592                     cdev = pdev;
593                     pdev = pdev->parent;
594                 }
595                 info("USB new device connect on bus%d/%s, assigned device number %d",
596                     dev->bus->busnum, portstr, dev->devnum);
597             } else
598                 info("USB new device connect on bus%d, assigned device number %d",
```

```
599                     dev->bus->busnum, dev->devnum);
600
601             /* Run it through the hoops (find a driver, etc) */
602             if (!usb_new_device(dev))
603                 goto done;
604
605             /* Free the configuration if there was an error */
606             usb_free_dev(dev);
607
608             /* Switch to a long reset time */
609             delay = HUB_LONG_RESET_TIME;
610         }
611
612         hub->children[port] = NULL;
613         usb_hub_port_disable(hub, port);
614     done:
615         up(&usb_address0_sem);
616         if (portstr)
617             kfree(portstr);
618         if (tempstr)
619             kfree(tempstr);
620     }
```

从设备读入的信息中包括两个 16 位状态标志字，一个表示端口当前的状态，一个表示哪些状态发生了变化。在 drivers/usb/hub.h 中定义了一些有关的常数：

```
40    /* wPortStatus bits */
41    #define USB_PORT_STAT_CONNECTION    0x0001
42    #define USB_PORT_STAT_ENABLE        0x0002
43    #define USB_PORT_STAT_SUSPEND       0x0004
44    #define USB_PORT_STAT_OVERCURRENT   0x0008
45    #define USB_PORT_STAT_RESET         0x0010
46    #define USB_PORT_STAT_POWER         0x0100
47    #define USB_PORT_STAT_LOW_SPEED     0x0200
48
49    /* wPortChange bits */
50    #define USB_PORT_STAT_C_CONNECTION  0x0001
51    #define USB_PORT_STAT_C_ENABLE      0x0002
52    #define USB_PORT_STAT_C_SUSPEND     0x0004
53    #define USB_PORT_STAT_C_OVERCURRENT 0x0008
54    #define USB_PORT_STAT_C_RESET       0x0010
```

对于连接状态发生了变化的端口，先通过 usb_clear_port_feature( )启动一次对目标集中器的控制传输，清除该端口的状态信息(drivers/usb/usb.c)。

[usb_hub_thread( ) > usb_hub_events( ) > usb_hub_port_connect_change( ) > usb_clear_port_feature ( )]

```
54    static int usb_clear_port_feature(struct usb_device *dev, int port, int feature)
55    {
56        return usb_control_msg(dev, usb_sndctrlpipe(dev, 0),
57            USB_REQ_CLEAR_FEATURE, USB_RT_PORT, feature, port, NULL, 0, HZ);
58    }
```

然后，如果 usb_device 数据结构中的 children[port]非 0，就要调用 usb_disconnect( )断开该端口上的连接，这个函数的代码在 drivers/usb/usb.c 中：

[usb_hub_thread( ) > usb_hub_events( ) > usb_hub_port_connect_change( ) > usb_disconnect( )]

```
1619    /*
1620     * Something got disconnected. Get rid of it, and all of its children.
1621     */
1622    void usb_disconnect(struct usb_device **pdev)
1623    {
1624        struct usb_device * dev = *pdev;
1625        int i;
1626
1627        if (!dev)
1628            return;
1629
1630        *pdev = NULL;
1631
1632        info("USB disconnect on device %d", dev->devnum);
1633
1634        if (dev->actconfig) {
1635            for (i = 0; i < dev->actconfig->bNumInterfaces; i++) {
1636                struct usb_interface *interface = &dev->actconfig->interface[i];
1637                struct usb_driver *driver = interface->driver;
1638                if (driver) {
1639                    down(&driver->serialize);
1640                    driver->disconnect(dev, interface->private_data);
1641                    up(&driver->serialize);
1642                    usb_driver_release_interface(driver, interface);
1643                }
1644            }
1645        }
1646
1647        /* Free up all the children.. */
1648        for (i = 0; i < USB_MAXCHILDREN; i++) {
1649            struct usb_device **child = dev->children + i;
1650            if (*child)
1651                usb_disconnect(child);
1652        }
1653
1654        /* Let policy agent unload modules etc */
```

```
1655        call_policy ("remove", dev);
1656
1657        /* Free the device number and remove the /proc/bus/usb entry */
1658        if (dev->devnum > 0) {
1659            clear_bit(dev->devnum, &dev->bus->devmap.devicemap);
1660            usbdevfs_remove_device(dev);
1661        }
1662
1663        /* Free up the device itself */
1664        usb_free_dev(dev);
1665    }
```

我们把这个函数留给读者自己阅读。

初看之下这似乎有点奇怪，我们并没有检查这个端口连接状态的变化到底是朝什么方向的变化，怎么能凭 hub->children[port]非 0 就决定断开它的连接呢(537 行)？其实这正是这段代码的精炼之处。如果连接状态的变化是从无到有，那么 hub->children[port]应该是 0，所以实际上不会调用这个函数。万一 hub->children[port]非 0，就说明这是“漏网之鱼”，所以应该将其清除。反过来，如果连接状态的变化是从有到无，那么这就更是我们所需要做的。

进一步，如果端口状态表明无连接（541 行），而端口又处于开通状态（542 行），那就要通过 usb_hub_port_disable( )启动另一次控制传输，将目标端口封闭，然后就返回了（545 行）。

如果 CPU 执行到了 548 行，那就说明所发生的变化是从无到有，也就是有个 USB 设备插入了这个端口。这以后的操作，如 usb_alloc_dev( )、usb_hub_port_reset( )、usb_connect( )和 usb_new_device( )等，就都是前面已经看到过的了，这就是对新设备的枚举过程。注意这里对 usb_new_device( )并非递归调用。如果新的设备又是个集中器，那么 usb_new_device( )会通过 usb_find_drivers( )找到集中器的驱动模块，并执行其 probe 函数。而集中器驱动模块的 probe 函数又会为新的集中器调度好中断交互，因而 khubd 又会对新的集中器执行 usb_hub_events( )。如果不是集中器呢，那就要看具体设备的驱动模块是否已经安装。如果该设备的驱动模块已经安装，那么 usb_new_device( )通过 usb_find_drivers( )同样会找到其驱动模块，并执行其 probe 函数。反之，如果该设备的驱动模块尚未安装，那就要到安装其驱动模块时再来认领并执行其 probe 函数。下面我们将结合扫描器的驱动说明这个过程。

回到 usb_hub_events( )的代码中，对每个端口还要检查和处理其他一些状态变化并作出处理。最后，还要检查整个集中器的电源状态并作出反应(720~735 行)。不过，那些就不是我们在这里所关心的了，有兴趣或需要的读者可自行阅读。

8.9.3 USB 设备的初始化

我们以扫描器为实例来说明一般 USB 设备的初始化过程。

首先，每种 USB 设备都要有个 usb_driver 数据结构，这是对 USB 设备的最高层次上的抽象。以我们在前一小节中看到 USB 集中器为例，其 usb_driver 数据结构是 hub_driver。这种数据结构的定义在 include/linux/usb.h 中：

```
384    struct usb_driver {
385        const char *name;
386
387        void *(*probe)(
388            struct usb_device *dev,     /* the device */
389            unsigned intf,          /* what interface */
390            const struct usb_device_id *id  /* from id_table */
391            );
392        void (*disconnect)(struct usb_device *, void *);
393
394        struct list_head driver_list;
395
396        struct file_operations *fops;
397        int minor;
398
399        struct semaphore serialize;
400
401        /* ioctl -- userspace apps can talk to drivers through usbdevfs */
402        int (*ioctl)(struct usb_device *dev, unsigned int code, void *buf);
403
404        /* support for "new-style" USB hotplugging
405         * binding policy can be driven from user mode too
406         */
407        const struct usb_device_id *id_table;
408
409        /* suspend before the bus suspends;
410         * disconnect or resume when the bus resumes */
411        // void (*suspend)(struct usb_device *dev);
412        // void (*resume)(struct usb_device *dev);
413    };
```

扫描器的 usb_driver 结构是 scanner_driver，定义于 drivers/usb/scanner.c：

```
953    static struct
954    usb_driver scanner_driver = {
955        name:       "usbscanner",
956        probe:      probe_scanner,
957        disconnect: disconnect_scanner,
958        fops:       &usb_scanner_fops,
959        minor:      SCN_BASE_MNR,
960        id_table:   NULL, /* This would be scanner_device_ids, but we
961                     need to check every USB device, in case
962                     we match a user defined vendor/product ID. */
963    };
```

结构中的成分 probe 和 disconnect 是两个函数指针，分别指向 probe_scanner( ) 和 disconnect_scanner( )。顾名思义，前者实际上用于扫描器的初始化，后者则用于断开与扫描器的连接。

指针 fops 则指向扫描器设备的 file_operations 数据结构 usb_scanner_fops，也定义于 drivers/usb/scanner.c：

```
942	static struct
943	file_operations usb_scanner_fops = {
944		read:		read_scanner,
945		write:		write_scanner,
946	#ifdef SCN_IOCTL
947		ioctl:		ioctl_scanner,
948	#endif /* SCN_IOCTL */
949		open:		open_scanner,
950		release:	close_scanner,
951	};
```

此外，scanner_driver 中还有个字段 minor，这是具体扫描器的次设备号，其初始值固定为 SCN_BASE_MNR。这个常数定义于 drivers/usb/scanner.h：

```
84	#define SCN_MAX_MNR 16		/* We're allocated 16 minors */
85	#define SCN_BASE_MNR 48		/* USB Scanners start at minor 48 */
```

就是说，扫描器的次设备号从 48 开始，系统中最多可以有 16 台扫描器。

像其他设备一样，作为 USB 设备的扫描器也需要先向系统登记，这就是扫描器驱动模块初始化过程要完成的操作。drivers/usb/scanner.c 中定义了扫描器的初始化函数 usb_scanner_init( )。

```
971	int __init
972	usb_scanner_init (void)
973	{
974		if (usb_register(&scanner_driver) < 0)
975			return -1;
976
977		info("USB Scanner support registered.");
978		return 0;
979	}
980
981	module_init(usb_scanner_init);
```

函数 usb_register( )的代码在 drivers/usb/usb.c 中：

[usb_scanner_init( ) > usb_register( )]

```
85	int usb_register(struct usb_driver *new_driver)
86	{
87		if (new_driver->fops != NULL) {
88			if (usb_minors[new_driver->minor/16]) {
89				 err("error registering %s driver", new_driver->name);
90				return -EINVAL;
91			}
```

```
 92             usb_minors[new_driver->minor/16] = new_driver;
 93         }
 94 
 95         info("registered new driver %s", new_driver->name);
 96 
 97         init_MUTEX(&new_driver->serialize);
 98 
 99         /* Add it to the list of known drivers */
100         list_add_tail(&new_driver->driver_list, &usb_driver_list);
101 
102         usb_scan_devices( );
103 
104         return 0;
105     }
```

内核中为 USB 设备设立了一个 usb_driver 结构指针数组 usb_minors[ ]，其大小为 16，每个具体 usb_driver 结构的次设备号除以 16 便决定了它在数组中的位置。每登记一个具体的 usb_driver 结构以后，数组中相应的指针就指向了这个数据结构。由此可见，USB 设备的次设备号实际上分成了两截，其高 4 位起着类似于主设备号的作用，而其低 4 位则是真正的“次设备号”。这样，系统中可以同时存在 16 种不同的 USB 设备，而每种设备(如扫描器)最多可以有 16 台。这当然只是权宜之计，将来随着 devfs 的采用应该会有更好的解决。

为了保证应用进程对 USB 设备操作的互斥性，usb_driver 数据结构中有个内核信号量 serialize，代码中对其进行了初始化。

内核中有两个 USB 层次上的队列。一个是 usb_bus 结构的队列 usb_bus_list，另一个是 usb_driver 结构的队列 usb_driver_list。所有 USB 设备驱动模块的 usb_driver 结构都链接在这个队列中，新驱动模块的 usb_driver 结构就挂在 usb_driver_list 的尾部。

然后，通过 usb_scan_devices( )扫描所有 USB 总线上的所有设备，让每个 USB 设备驱动模块都试着来“认领”与其对口的设备。就扫描器的驱动模块而言，就是要让它认领已经枚举的扫描器设备。函数 usb_scan_devices( )的代码在 drivers/usb/usb.c 中：

[usb_scanner_init( ) > usb_register( ) > usb_scan_devices( )]

```
107     /**
108      * usb_scan_devices - scans all unclaimed USB interfaces
109      *
110      * Goes through all unclaimed USB interfaces, and offers them to all
111      * registered USB drivers through the 'probe' function.
112      * This will automatically be called after usb_register is called.
113      * It is called by some of the USB subsystems after one of their subdrivers
114      * are registered.
115      */
116     void usb_scan_devices(void)
117     {
118         struct list_head *tmp;
```

```
119
120         tmp = usb_bus_list.next;
121         while (tmp != &usb_bus_list) {
122             struct usb_bus *bus = list_entry(tmp, struct usb_bus, bus_list);
123
124             tmp = tmp->next;
125             usb_check_support(bus->root_hub);
126         }
127     }
```

这个函数扫描队列 usb_bus_list 中的每一个 usb_bus 数据结构，也就是系统中的每一条 USB 总线，通过一个函数 usb_check_support( )检查该总线上已经枚举的设备。其代码在 drivers/usb/usb.c 中：

[usb_scanner_init( ) > usb_register( ) > usb_scan_devices( ) > usb_check_support( )]

```
434     /*
435      * This function is for doing a depth-first search for devices which
436      * have support, for dynamic loading of driver modules.
437      */
438     static void usb_check_support(struct usb_device *dev)
439     {
440         int i;
441
442         if (!dev) {
443             err("null device being checked!!!");
444             return;
445         }
446
447         for (i=0; i<USB_MAXCHILDREN; i++)
448             if (dev->children[i])
449                 usb_check_support(dev->children[i]);
450
451         if (!dev->actconfig)
452             return;
453
454         /* now we check this device */
455         if (dev->devnum > 0)
456             for (i = 0; i < dev->actconfig->bNumInterfaces; i++)
457                 usb_find_interface_driver(dev, i);
458     }
```

每条 USB 总线的“根”是一个 USB 集中器，那就是根集中器。主机的 USB 控制器一般都与根集中器集成在一起。根集中器也是由 usb_device 数据结构代表的，在 USB 总线初始化时建立起它的数据结构。每个 USB 集中器可以连接若干 USB 设备，其数量取决于具体的集中器，但是最多不超过 USB_MAXCHILDREN，目前这个常数定义为 16。连接在 USB 集中器上的 USB 设备本身又可以是 USB 集中器，所以代码中对下接的集中器递归调用 usb_check_support( )，进行深度优先的搜索，一直到不

再是集中器，或者尚未插上设备的集中器时才回头。

如果一个 USB 设备已经分配了地址并已成功枚举，也没有被“除名”，其 usb_device 结构中的字段 devnum 就是一个正数，此时结构中的指针 actconfig 指向一个 usb_config_descriptor 数据结构。我们已经在前几个小节中看到这种数据结构的定义。

这个结构中的字段 bNumInterfaces 表示相应的 USB 设备中有着几个逻辑设备，从而有几个“接口”。指针 interface 指向一个 usb_interface 数据结构数组，其定义也已在前几个小节中看到过，数组的大小则取决于 bNumInterfaces。

每一个接口可以包含若干“端点”(endpoint)，对于主机来说，每个端点就是一个通信的对象。不管是什么样的 USB 设备，至少要有一个公用的端点用于控制以及枚举的目的，这个端点的“端点号”固定为 0。除此之外，每一个接口还可以有附加的、专有的端点。端点号 0 所代表的通信对象是双向的，既可以收也可以发，而其他端点号所代表的通信对象则都是单向的。这个数据结构中的字段 bNumEndpoints 说明了本接口中有几个端点，而指针 endpoint 则指向一个 usb_endpoint_descriptor 结构数组，每个 usb_endpoint_descriptor 都代表着一个端点，其定义见前一小节。

从代表着具体 USB 总线的 usb_bus 结构到代表着具体设备(包括集中器)的 usb_device 结构，再到 usb_config_descriptor 结构，最后是 usb_interface 结构和 usb_endpoint_descriptor 结构，正好反映了 USB 的各个结构层次。这些数据结构本身以及互相间的联系，都是在 USB 总线初始化时，或者后来当 USB 总线上的设备连接有变化时的枚举阶段建立的。枚举阶段所获取的这些信息反映了当前 USB 总线上各个设备(端点)的客观存在，但是尚未与具体的设备驱动程序挂上钩。所以，在具体设备(类型)的初始化中要在这些数据结构中找到对口的数据结构并加以“认领”(claim)。

函数 usb_find_interface_driver( )逐个地检查数组中的每个 usb_interface 数据结构，看看相应逻辑设备的类型是否相符，如果相符就加以初始化。其代码在 drivers/usb/usb.c 中：

[usb_scanner_init( ) > usb_register( ) > usb_scan_devices( ) > usb_check_support( ) > usb_find_interface_driver( )]

```
623    /*
624     * This entrypoint gets called for each new device.
625     *
626     * We now walk the list of registered USB drivers,
627     * looking for one that will accept this interface.
628     *
629     * "New Style" drivers use a table describing the devices and interfaces
630     * they handle.  Those tables are available to user mode tools deciding
631     * whether to load driver modules for a new device.
632     *
633     * The probe return value is changed to be a private pointer.  This way
634     * the drivers don't have to dig around in our structures to set the
635     * private pointer if they only need one interface.
636     *
637     * Returns: 0 if a driver accepted the interface, -1 otherwise
638     */
639    static int usb_find_interface_driver(struct usb_device *dev, unsigned ifnum)
```

```
640    {
641        struct list_head *tmp;
642        struct usb_interface *interface;
643        void *private;
644        const struct usb_device_id *id;
645        struct usb_driver *driver;
646        int i;
647
648        if ((!dev) || (ifnum >= dev->actconfig->bNumInterfaces)) {
649            err("bad find_interface_driver params");
650            return -1;
651        }
652
653        interface = dev->actconfig->interface + ifnum;
654
655        if (usb_interface_claimed(interface))
656            return -1;
657
658        private = NULL;
659        for (tmp = usb_driver_list.next; tmp != &usb_driver_list;) {
660
661            driver = list_entry(tmp, struct usb_driver, driver_list);
662            tmp = tmp->next;
663
664            down(&driver->serialize);
665            id = driver->id_table;
666            /* new style driver? */
667            if (id) {
668                for (i = 0; i < interface->num_altsetting; i++) {
669                    interface->act_altsetting = i;
670                    id = usb_match_id(dev, interface, id);
671                    if (id) {
672                        private = driver->probe(dev,ifnum,id);
673                        if (private != NULL)
674                            break;
675                    }
676                }
677                /* if driver not bound, leave defaults unchanged */
678                if (private == NULL)
679                    interface->act_altsetting = 0;
680            }
681            else /* "old style" driver */
682                private = driver->probe(dev, ifnum, NULL);
683
684            up(&driver->serialize);
685            if (private) {
686                usb_driver_claim_interface(driver, interface, private);
687                return 0;
```

```
688                 }
689             }
690
691             return -1;
692     }
```

这里的参数 dev 指向一个 USB 设备的 usb_device 数据结构，以接口号为下标，就可以在其数组 interface[ ]中找到具体接口，即逻辑设备的 usb_interface 结构。如果这个结构中的指针 driver 已经指向一个 usb_driver 数据结构，则该接口已被认领。否则，就在 usb_driver_list 队列中扫描已经登记的 usb_driver 数据结构，逐个地通过 usb_match_id( )将其 id 表跟设备上的每个接口比对。这个函数的代码在 drivers/usb/usb.c 中：

[usb_scanner_init( ) > usb_register( ) > usb_scan_devices( ) > usb_check_support( )
 > usb_find_interface_driver( ) > usb_match_id( )]

```
505     /* usb_match_id searches an array of usb_device_id's and returns
506        the first one that matches the device and interface.
507
508        Parameters:
509         "id" is an array of usb_device_id's is terminated by an entry
510          containing all zeroes.
511
512          "dev" and "interface" are the device and interface for which
513          a match is sought.
514
515        If no match is found or if the "id" pointer is NULL, then
516        usb_match_id returns NULL.
517
518
519        What constitutes a match:
520
521        A zero in any element of a usb_device_id entry is a wildcard
522        (i.e., that field always matches).  For there to be a match,
523        *every* nonzero element of the usb_device_id must match the
524        provided device and interface in.  The comparison is for equality,
525        except for one pair of fields: usb_match_id.bcdDevice_{lo,hi} define
526        an inclusive range that dev->descriptor.bcdDevice must be in.
527
528        If interface->altsettings does not exist (i.e., there are no
529        interfaces defined), then bInterface{Class,SubClass,Protocol}
530        only match if they are all zeroes.
531
532
533        What constitutes a good "usb_device_id"?
534
535        The match algorithm is very simple, so that intelligence in
```

```
536        driver selection must come from smart driver id records.
537        Unless you have good reasons to use another selection policy,
538        provide match elements only in related groups:
539
540         * device specifiers (vendor and product IDs; and maybe
541           a revision range for that product);
542         * generic device specs (class/subclass/protocol);
543         * interface specs (class/subclass/protocol).
544
545        Within those groups, work from least specific to most specific.
546        For example, don't give a product version range without vendor
547        and product IDs.
548
549        "driver_info" is not considered by the kernel matching algorithm,
550        but you can create a wildcard "matches anything" usb_device_id
551        as your driver's "modules.usbmap" entry if you provide only an
552        id with a nonzero "driver_info" field.
553     */
554
555     const struct usb_device_id *
556     usb_match_id(struct usb_device *dev, struct usb_interface *interface,
557              const struct usb_device_id *id)
558     {
559         struct usb_interface_descriptor *intf = 0;
560
561         /* proc_connectinfo in devio.c may call us with id == NULL. */
562         if (id == NULL)
563             return NULL;
564
565         /* It is important to check that id->driver_info is nonzero,
566            since an entry that is all zeroes except for a nonzero
567            id->driver_info is the way to create an entry that
568            indicates that the driver want to examine every
569            device and interface. */
570         for (; id->idVendor || id->bDeviceClass || id->bInterfaceClass ||
571                id->driver_info; id++) {
572
573             if ((id->match_flags & USB_DEVICE_ID_MATCH_VENDOR) &&
574                 id->idVendor != dev->descriptor.idVendor)
575                 continue;
576
577             if ((id->match_flags & USB_DEVICE_ID_MATCH_PRODUCT) &&
578                 id->idProduct != dev->descriptor.idProduct)
579                 continue;
580
581             /* No need to test id->bcdDevice_lo != 0, since 0 is never
582                greater than any unsigned number. */
583             if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_LO) &&
```

```
584                 (id->bcdDevice_lo > dev->descriptor.bcdDevice))
585                 continue;
586 
587             if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_HI) &&
588                 (id->bcdDevice_hi < dev->descriptor.bcdDevice))
589                 continue;
590 
591             if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_CLASS) &&
592                 (id->bDeviceClass != dev->descriptor.bDeviceClass))
593                 continue;
594 
595             if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_SUBCLASS) &&
596                 (id->bDeviceSubClass!= dev->descriptor.bDeviceSubClass))
597                 continue;
598 
599             if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_PROTOCOL) &&
600                 (id->bDeviceProtocol != dev->descriptor.bDeviceProtocol))
601                 continue;
602 
603             intf = &interface->altsetting [interface->act_altsetting];
604 
605             if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_CLASS) &&
606                 (id->bInterfaceClass != intf->bInterfaceClass))
607                 continue;
608 
609             if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_SUBCLASS) &&
610                 (id->bInterfaceSubClass != intf->bInterfaceSubClass))
611                 continue;
612 
613             if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_PROTOCOL) &&
614                 (id->bInterfaceProtocol != intf->bInterfaceProtocol))
615                 continue;
616 
617             return id;
618         }
619 
620         return NULL;
621     }
```

登记的每个 usb_driver 结构都应该有个 id 表，说明该驱动程序（模块）适用于哪一些设备。所谓 id 表是个 usb_device_id 结构数组，这种数据结构定义于 include/linux/usb.h：

```
347     struct usb_device_id {
348         /* This bitmask is used to determine which of the following fields
349          * are to be used for matching.
350          */
351         __u16       match_flags;
```

```
352
353         /*
354          * vendor/product codes are checked, if vendor is nonzero
355          * Range is for device revision (bcdDevice), inclusive;
356          * zero values here mean range isn't considered
357          */
358         __u16       idVendor;
359         __u16       idProduct;
360         __u16       bcdDevice_lo, bcdDevice_hi;
361
362         /*
363          * if device class != 0, these can be match criteria;
364          * but only if this bDeviceClass value is nonzero
365          */
366         __u8        bDeviceClass;
367         __u8        bDeviceSubClass;
368         __u8        bDeviceProtocol;
369
370         /*
371          * if interface class != 0, these can be match criteria;
372          * but only if this bInterfaceClass value is nonzero
373          */
374         __u8        bInterfaceClass;
375         __u8        bInterfaceSubClass;
376         __u8        bInterfaceProtocol;
377
378         /*
379          * for driver's use; not involved in driver matching.
380          */
381         unsigned long   driver_info;
382     };
```

每个 usb_device_id 结构描述着一个驱动程序可以适用的对象，包括由谁制造、制造商的产品编号、设备的类型、接口的类型等等特征信息。同时，结构中还有个位图 match_flags，说明必须有哪些特征信息都相符才可以把一个设备及接口认定为与给定驱动程序相符。如果 usb_match_id( )返回非 0，就说明找到了适用于目标设备的驱动程序。然后，就可以让这个驱动模块的 probe 函数来进行该设备的初始化了。对于扫描器，其 probe 函数为 probe_scanner( )，它的代码在 drivers/usb/scanner.c 中。我们分段阅读。

[usb_scanner_init( ) > usb_register( ) > usb_scan_devices( ) > usb_check_support( ) > usb_find_interface_driver( ) > probe_scanner( )]

```
626     static void *
627     probe_scanner(struct usb_device *dev, unsigned int ifnum,
628                 const struct usb_device_id *id)
629     {
```

```
630         struct scn_usb_data *scn;
631         struct usb_interface_descriptor *interface;
632         struct usb_endpoint_descriptor *endpoint;
633 
634         int ep_cnt;
635         int ix;
636 
637         kdev_t scn_minor;
638 
639         char valid_device = 0;
640         char have_bulk_in, have_bulk_out, have_intr;
641 
642         if (vendor != -1 && product != -1) {
643             info(
        "probe_scanner: User specified USB scanner -- Vendor:Product - %x:%x",
                                                     vendor, product);
644         }
645 
646         dbg("probe_scanner: USB dev address:%p", dev);
647         dbg("probe_scanner: ifnum:%u", ifnum);
648 
649     /*
650      * 1. Check Vendor/Product
651      * 2. Determine/Assign Bulk Endpoints
652      * 3. Determine/Assign Intr Endpoint
653      */
654 
655     /*
656      * There doesn't seem to be an imaging class defined in the USB
657      * Spec. (yet).  If there is, HP isn't following it and it doesn't
658      * look like anybody else is either.  Therefore, we have to test the
659      * Vendor and Product ID's to see what we have.  Also, other scanners
660      * may be able to use this driver by specifying both vendor and
661      * product ID's as options to the scanner module in conf.modules.
662      *
663      * NOTE: Just because a product is supported here does not mean that
664      * applications exist that support the product.  It's in the hopes
665      * that this will allow developers a means to produce applications
666      * that will support USB products.
667      *
668      * Until we detect a device which is pleasing, we silently punt.
669      */
670 
671         for (ix = 0;
                ix < sizeof(scanner_device_ids) / sizeof(struct usb_device_id);
                ix++) {
672             if ((dev->descriptor.idVendor == scanner_device_ids [ix].idVendor) &&
673                 (dev->descriptor.idProduct == scanner_device_ids [ix].idProduct)) {
```

```
674                 valid_device = 1;
675                 break;
676                     }
677         }
678         if (dev->descriptor.idVendor == vendor &&    /* User specified */
679             dev->descriptor.idProduct == product) { /* User specified */
680             valid_device = 1;
681         }
682
683             if (!valid_device)
684                     return NULL;    /* We didn't find anything pleasing */
685
686     /*
687      * After this point we can be a little noisy about what we are trying to
688      *  configure.
689      */
690
691         if (dev->descriptor.bNumConfigurations != 1) {
692             info("probe_scanner: Only one device configuration is supported.");
693             return NULL;
694         }
695
696         if (dev->config[0].bNumInterfaces != 1) {
697             info("probe_scanner: Only one device interface is supported.");
698             return NULL;
699         }
700
701         interface = dev->config[0].interface[ifnum].altsetting;
702         endpoint = interface[ifnum].endpoint;
703
```

首先将目标设备跟扫描器的 id 表 scanner_device_ids 加以核对。然后，使指针 interface 指向目标接口的 usb_interface_descriptor 数据结构。这个结构中的指针 endpoint 指向一个端点描述结构数组，结构中的另一个字段 bNumEndpoints 则记录着这个数组的大小，即本接口的端点数量。数组中的每一个 usb_endpoint_descriptor 结构都代表着目标设备中的一个端点。下面，就要对这些端点进行验证，我们继续往下看 probe_scanner( )的代码(drivers/usb/scanner.c)。

[usb_scanner_init( ) > usb_register( ) > usb_scan_devices( ) > usb_check_support( )
 > usb_find_interface_driver( ) > probe_scanner( )]

```
704     /*
705      * Start checking for two bulk endpoints OR two bulk endpoints *and* one
706      * interrupt endpoint. If we have an interrupt endpoint go ahead and
707      * setup the handler. FIXME: This is a future enhancement...
708      */
709
```

```
710        dbg("probe_scanner: Number of Endpoints:%d", (int) interface->bNumEndpoints);
711
712        if ((interface->bNumEndpoints != 2) && (interface->bNumEndpoints != 3)) {
713            info("probe_scanner: Only two or three endpoints supported.");
714            return NULL;
715        }
716
717        ep_cnt = have_bulk_in = have_bulk_out = have_intr = 0;
718
719        while (ep_cnt < interface->bNumEndpoints) {
720
721            if (!have_bulk_in && IS_EP_BULK_IN(endpoint[ep_cnt])) {
722                ep_cnt++;
723                have_bulk_in = ep_cnt;
724                dbg("probe_scanner: bulk in_ep:%d", have_bulk_in);
725                continue;
726            }
727
728            if (!have_bulk_out && IS_EP_BULK_OUT(endpoint[ep_cnt])) {
729                ep_cnt++;
730                have_bulk_out = ep_cnt;
731                dbg("probe_scanner: bulk_out_ep:%d", have_bulk_out);
732                continue;
733            }
734
735            if (!have_intr && IS_EP_INTR(endpoint[ep_cnt])) {
736                ep_cnt++;
737                have_intr = ep_cnt;
738                dbg("probe_scanner: intr_ep:%d", have_intr);
739                continue;
740            }
741            info("probe_scanner: Undetected endpoint. Notify the maintainer.");
742            return NULL;/* Shouldn't ever get here unless we have something weird */
743        }
744
745
746    /*
747     * Perform a quick check to make sure that everything worked as it
748     * should have.
749     */
750
751        switch(interface->bNumEndpoints) {
752        case 2:
753            if (!have_bulk_in || !have_bulk_out) {
754                info("probe_scanner: Two bulk endpoints required.");
755                return NULL;
756            }
757            break;
```

```
758         case 3:
759             if (!have_bulk_in || !have_bulk_out || !have_intr) {
760                 info("probe_scanner: \
                        Two bulk endpoints and one interrupt endpoint required.");
761                 return NULL;
762             }
763             break;
764         default:
765             info("probe scanner: \
                        Endpoint determination failed.  Notify the maintainer.");
766             return NULL;
767         }
768
769
```

除控制端点外，扫描器的接口上还应该有输出和输入两个成块传输端点。此外可能还有一个中断传输端点。这里用到的几个宏操作均定义于 drivers/usb/scanner.h 中：

```
59  #define IS_EP_BULK(ep)  ((ep).bmAttributes == USB_ENDPOINT_XFER_BULK ? 1 : 0)
60  #define IS_EP_BULK_IN(ep)  (IS_EP_BULK(ep) && \
                    ((ep).bEndpointAddress & USB_ENDPOINT_DIR_MASK) == USB_DIR_IN)
61  #define IS_EP_BULK_OUT(ep)  (IS_EP_BULK(ep) && \
                    ((ep).bEndpointAddress & USB_ENDPOINT_DIR_MASK) == USB_DIR_OUT)
62  #define IS_EP_INTR(ep)  ((ep).bmAttributes == USB_ENDPOINT_XFER_INT ? 1 : 0)
```

每个端点都有个单字节的地址，其低 4 位为端点号，最高位则指明其方向，所以检查端点地址的最高位就可以知道块状端点的方向为输出或输入。当然，这个方向位对于双向的端点没有意义。此外，端点还有个属性字节，其最低两位表明端点的类型。有关的常数均定义于 include/linux/usb.h:

```
72  #define USB_ENDPOINT_NUMBER_MASK    0x0f    /* in bEndpointAddress */
73  #define USB_ENDPOINT_DIR_MASK       0x80
74
75  #define USB_ENDPOINT_XFERTYPE_MASK  0x03    /* in bmAttributes */
76  #define USB_ENDPOINT_XFER_CONTROL   0
77  #define USB_ENDPOINT_XFER_ISOC      1
78  #define USB_ENDPOINT_XFER_BULK      2
79  #define USB_ENDPOINT_XFER_INT       3
```

确认了扫描器的接口具有适当数量的端点以后，要为这个扫描器分配一个次设备号和一个 scn_usb_data 数据结构。这种数据结构定义于 drivers/usb/scanner.h:

```
87  struct scn_usb_data {
88      struct usb_device *scn_dev;
89      struct urb scn_irq;
90      unsigned int ifnum; /* Interface number of the USB device */
91      kdev_t scn_minor;   /* Scanner minor - used in disconnect( ) */
```

```
92        unsigned char button;   /* Front panel buffer */
93        char isopen;         /* Not zero if the device is open */
94        char present;        /* Not zero if device is present */
95        char *obuf, *ibuf;  /* transfer buffers */
96        char bulk_in_ep, bulk_out_ep, intr_ep; /* Endpoint assignments */
97        wait_queue_head_t rd_wait_q; /* read timeouts */
98        struct semaphore gen_lock; /* lock to prevent concurrent reads or writes */
99    };
```

结构中的指针 obuf 和 ibuf 分别指向用于该扫描器的输出和输入缓冲区，所以要为之分配空间。

[usb_scanner_init( ) > usb_register( ) > usb_scan_devices( ) > usb_check_support( ) > usb_find_interface_driver( ) > probe_scanner( )]

```
770    /*
771     * Determine a minor number and initialize the structure associated
772     * with it.  The problem with this is that we are counting on the fact
773     * that the user will sequentially add device nodes for the scanner
774     * devices.  */
775
776        for (scn_minor = 0; scn_minor < SCN_MAX_MNR; scn_minor++) {
777            if (!p_scn_table[scn_minor])
778                break;
779        }
780
781    /* Check to make sure that the last slot isn't already taken */
782        if (p_scn_table[scn_minor]) {
783            err("probe_scanner: No more minor devices remaining.");
784            return NULL;
785        }
786
787        dbg("probe_scanner: Allocated minor:%d", scn minor);
788
789        if (!(scn = kmalloc (sizeof (struct scn usb data), GFP KERNEL))) {
790            err("probe_scanner: Out of memory.");
791            return NULL;
792        }
793        memset (scn, 0, sizeof(struct scn_usb_data));
794        dbg ("probe scanner(%d): Address of scn:%p", scn_minor, scn);
795
796
```

其他字段也需要根据已有的信息加以设置。特别地，如果设备具有中断端点，则要为这个端点调度一个周期性的中断传输，建立起对扫描器的定期查询(drivers/usb/scanner.c)。

[usb_scanner_init( ) > usb_register( ) > usb_scan_devices( ) > usb_check_support( ) > usb_find_interface_driver( ) > probe_scanner( )]

```
797     /* Ok, if we detected an interrupt EP, setup a handler for it */
798         if (have_intr) {
799             dbg("probe_scanner(%d): Configuring IRQ handler for intr EP:%d",
                                                        scn_minor, have_intr);
800             FILL_INT_URB(&scn->scn_irq, dev,
801                     usb_rcvintpipe(dev, have_intr),
802                     &scn->button, 1, irq_scanner, scn,
803                     // endpoint[(int)have_intr].bInterval);
804                     250);
805
806                 if (usb_submit_urb(&scn->scn_irq)) {
807                 err("probe_scanner(%d): Unable to allocate INT URB.", scn_minor);
808                         kfree(scn);
809                         return NULL;
810                 }
811         }
812
813
814     /* Ok, now initialize all the relevant values */
815         if (!(scn->obuf = (char *)kmalloc(OBUF_SIZE, GFP_KERNEL))) {
816             err("probe_scanner(%d): Not enough memory for the output buffer.",
                                                                    scn_minor);
817             kfree(scn);
818             return NULL;
819         }
820         dbg("probe_scanner(%d): obuf address:%p", scn_minor, scn->obuf);
821
822         if (!(scn->ibuf = (char *)kmalloc(IBUF_SIZE, GFP_KERNEL))) {
823             err("probe_scanner(%d): Not enough memory for the input buffer.",
                                                                    scn_minor);
824             kfree(scn->obuf);
825             kfree(scn);
826             return NULL;
827         }
828         dbg("probe_scanner(%d): ibuf address:%p", scn_minor, scn->ibuf);
829
830         scn->bulk_in_ep = have_bulk_in;
831         scn->bulk_out_ep = have_bulk_out;
832         scn->intr_ep = have_intr;
833         scn->present = 1;
834         scn->scn_dev = dev;
835         scn->scn_minor = scn_minor;
836         scn->isopen = 0;
837
838         init_MUTEX(&(scn->gen_lock));
839
840         return p_scn_table[scn_minor] = scn;
841     }
```

我们把为扫描器调度中断传输的过程搁一下。暂且假定 probe_scanner( )的操作已经完成，建立了扫描器的 usb_interface_descriptor 数据结构，并为之建立了 scn_usb_data 数据结构。回到 usb_find_interface_driver( )的代码中，下一步是通过 usb_driver_claim_interface( )正式“认领”这个设备(接口) (drivers/usb/usb.c)。

[usb_scanner_init( ) > usb_register( ) > usb_scan_devices( ) > usb_check_support( )
 > usb_find_interface_driver( ) > usb_driver_claim_interface( )]

```
461    /*
462     * This is intended to be used by usb device drivers that need to
463     * claim more than one interface on a device at once when probing
464     * (audio and acm are good examples).  No device driver should have
465     * to mess with the internal usb_interface or usb_device structure
466     * members.
467     */
468    void usb_driver_claim_interface(struct usb_driver *driver,
                                       struct usb_interface *iface, void* priv)
469    {
470        if (!iface || !driver)
471            return;
472
473        dbg("%s driver claimed interface %p", driver->name, iface);
474
475        iface->driver = driver;
476        iface->private_data = priv;
477    } /* usb_driver_claim_interface( ) */
```

总之，所谓“认领”就是使一个 usb_interface_descriptor 数据结构与相应设备的 usb_driver 结构挂上钩。这样，从具体设备的数据结构出发，就可以找到其设备驱动程序了。就这样，当 usb_scan_devices( )完成了对所有 USB 设备的扫描时，对扫描器设备的登记和初始化就完成了。最后，以次设备号中的低 4 位为下标的指针数组 p_scn_table[ ]纪录着指向每个具体 scn_usb_data 结构的地址。

8.9.4 USB 设备的驱动

要对一台扫描器操作时，首先也要通过系统调用 open( )打开设备文件。所有的 USB 设备都有相同的主设备号 USB_MAJOR，而根据次设备号划分具体的设备及其类型。所以，根据设备文件节点提供的主设备号，CPU 首先找到 USB 总线的 file_operations 数据结构 usb_fops，从中得到用于 open 操作的函数指针，这个指针指向 usb_open( )。其代码在 drivers/usb/usb.c 中：

```
2177   static int usb_open(struct inode * inode, struct file * file)
2178   {
2179       int minor = MINOR(inode->i_rdev);
2180       struct usb_driver *c = usb_minors[minor/16];
```

```
2181        int err = -ENODEV;
2182        struct file_operations *old_fops, *new_fops = NULL;
2183
2184        /*
2185         * No load-on-demand? Randy, could you ACK that it's really not
2186         * supposed to be done?                    -- AV
2187         */
2188        if (!c || !(new_fops = fops_get(c->fops)))
2189            return err;
2190        old_fops = file->f_op;
2191        file->f_op = new_fops;
2192        /* Curiouser and curiouser... NULL ->open( ) as "no device" ? */
2193        if (file->f_op->open)
2194            err = file->f_op->open(inode,file);
2195        if (err) {
2196            fops_put(file->f_op);
2197            file->f_op = fops_get(old_fops);
2198        }
2199        fops_put(old_fops);
2200        return err;
2201    }
```

然后，进一步根据次设备号从指针数组 usb_minors[ ]中找到扫描器的 usb_driver 结构。我们知道，file 结构中的指针 f_op 应该指出向目标设备的 file_operations 数据结构，2188 行引用的 fops_get( )是定义于 include/linux/fs.h 的宏操作：

```
860    #define fops_get(fops) \
861        (((fops) && (fops)->owner)   \
862            ? ( try_inc_mod_count((fops)->owner) ? (fops) : NULL ) \
863            : (fops))
```

扫描器的 file_operations 结构是 usb_scanner_fops，其函数指针 open 指向 open_scanner( )，这个函数的代码在 drivers/usb/scanner.c 中：

[usb_open( ) > open_scanner( )]

```
345    static int
346    open_scanner(struct inode * inode, struct file * file)
347    {
348        struct scn_usb_data *scn;
349        struct usb_device *dev;
350
351        kdev_t scn_minor;
352
353        int err=0;
354
```

```
355         lock_kernel( );
356
357         scn_minor = USB_SCN_MINOR(inode);
358
359         dbg("open_scanner: scn_minor:%d", scn_minor);
360
361         if (!p_scn_table[scn_minor]) {
362             err("open_scanner(%d): Unable to access minor data", scn_minor);
363             err = -ENODEV;
364             goto out_error;
365         }
366
367         scn = p_scn_table[scn_minor];
368
369         dev = scn->scn_dev;
370
371         if (!dev) {
372             err("open_scanner(%d): Scanner device not present", scn_minor);
373             err = -ENODEV;
374             goto out_error;
375         }
376
377         if (!scn->present) {
378             err("open_scanner(%d): Scanner is not present", scn_minor);
379             err = -ENODEV;
380             goto out_error;
381         }
382
383         if (scn->isopen) {
384             err("open_scanner(%d): Scanner device is already open", scn_minor);
385             err = -EBUSY;
386             goto out_error;
387         }
388
389         init_waitqueue_head(&scn->rd_wait_q);
390
391         scn->isopen = 1;
392
393         file->private_data = scn; /* Used by the read and write metheds */
394
395         MOD_INC_USE_COUNT;
396
397     out_error:
398
399         unlock_kernel( );
400
401         return err;
402     }
```

我们把这个函数留给读者。有关 p_scn_table[ ]的作用可参阅前一小节中 probe_scanner()的代码。

打开了扫描器设备，就可以对其进行读/写，从此以后的操作有着两个层次上的内容。其一是取决于具体设备，即扫描器本身的操作，例如“开启光源”、“设置对比度”、“设置扫描精度”、“开始扫描”等等；其二是 USB 总线的操作，即如何实现主机与目标设备之间的信息传输，使对具体设备的操作得以实现。如果拿内核与应用软件之间的关系作一个比拟，则前者可以比作应用软件，而后者可以比作内核。前者是目的，后者是手段和基础设施。就一般的设备而言，这二者都是在内核中实现的，对 USB 设备的驱动当然也可以把这二者都放在内核中实现。可是，对一些速度要求不高、操作不很频繁、与内核中其他成分关系又不大的设备，把前者放在用户空间也有好处。拿扫描器来说，不同厂商、不同型号的扫描器在具体的操作上可能有相当的区别，放在用户空间中实现可以提供更大的灵活性。所以，对扫描器本身的操作是在进程这一层次上实现的，而内核只提供对扫描器操作的手段。我们在这里的目的并不在于扫描器本身，而在于驱动 USB 设备的手段及基础设施，就好像本书的题材不在于各种应用软件，而在于使这些应用软件得以运行的手段及基础设施一样。

一般来说，从系统的角度来看，对设备的操作无非是两方面的信息交换，一方面是控制/状态信息的传输，另一方面是数据的传输，常常分别称为“控制通道”(control path)和“数据通道”(data path)。至于以何种方式实现这两种通道，则存在着一些灵活性。例如，对于扫描器的数据通道一般都是通过“成块”(bulk)方式传输的。这是因为对扫描器的输入显然没有实时要求，不需要通过“等时”(isochronous)方式传输。对于控制通道则更无采用等时传输的理由。但是，在采用控制传输还是成块传输方式来传递控制/状态信息这一点上，如果具体的设备允许并提供了相应的手段，却还是可以推敲的。扫描器就是这样，现代的扫描器大多支持“扫描器语言”，使得通过数据通道也可以实现控制目的(类似于终端设备的 Escape 序列)。如前所述，控制传输的优先级别比成块传输高，而且在 USB 总线上保留了一定的带宽用于控制传输，所以比较可靠。但是，若采用成块方式传输则可以简化程序设计，因为那样就可以不加区分地一律采用成块方式传输。从表面上看，这样会使控制/状态信息的传输降低优先级别并得不到保障，可是对于扫描器这样的设备其实并无多大区别。试想，如果 USB 总线的负荷已经大到不能保证传输数据的地步，那么提高控制/状态信息的优先级别，以此来保证能启动扫描还有多大意义？所以，在扫描器的设备驱动代码中有个条件编译控制 SCN_IOCTL。如果选择了这个选项，则应用进程须通过系统调用 ioctl( )传递控制/状态信息，而通过 read( )/write( )读/写数据；否则便一律通过 read( )/write( )对扫描器操作。我们在这里假定选择通过 ioctl( )传递控制/状态信息，目的是让读者由此可以看到控制传输的实例。扫描器的 ioctl( )函数为 ioctl_scanner( )。其代码在 drivers/usb/scanner.c 中：

```
863     #ifdef SCN_IOCTL
864     static int
865     ioctl_scanner(struct inode *inode, struct file *file,
866                unsigned int cmd, unsigned long arg)
867     {
868         struct usb_device *dev;
869
870         int result;
871
872         kdev_t scn_minor;
873
874         scn_minor = USB_SCN_MINOR(inode);
```

```
875
876         if (!p_scn_table[scn_minor]) {
877             err("ioctl_scanner(%d): invalid scn_minor", scn_minor);
878             return -ENODEV;
879         }
880
881         dev = p_scn_table[scn_minor]->scn_dev;
882
883         switch (cmd)
884         {
885         case PV8630_IOCTL_INREQUEST :
886         {
887             struct {
888                 __u8  data;
889                 __u8  request;
890                 __u16 value;
891                 __u16 index;
892             } args;
893
894             if (copy_from_user(&args, (void *)arg, sizeof(args)))
895                 return -EFAULT;
896
897             result = usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
898                         args.request, USB_TYPE_VENDOR|
899                         USB_RECIP_DEVICE|USB_DIR_IN,
900                         args.value, args.index, &args.data,
901                         1, HZ*5);
902
903             dbg("ioctl_scanner(%d): inreq: args.data:%x args.value:\
                 %x args.index:%x args.request:%x\n",
                    scn_minor, args.data, args.value, args.index, args.request);
904
905             if (copy_to_user((void *)arg, &args, sizeof(args)))
906                 return -EFAULT;
907
908             dbg("ioctl_scanner(%d): inreq: result:%d\n", scn_minor, result);
909
910             return result;
911         }
912         case PV8630_IOCTL_OUTREQUEST :
913         {
914             struct {
915                 __u8  request;
916                 __u16 value;
917                 __u16 index;
918             } args;
919
920             if (copy_from_user(&args, (void *)arg, sizeof(args)))
```

```
921                 return -EFAULT;
922 
923             dbg("ioctl_scanner(%d): outreq: args.value:%x args.index:\
                    %x args.request:%x\n",
                            scn_minor, args.value, args.index, args.request);
924 
925             result = usb_control_msg(dev, usb_sndctrlpipe(dev, 0),
926                             args.request, USB_TYPE_VENDOR|
927                             USB_RECIP_DEVICE|USB DIR OUT,
928                             args.value, args.index, NULL,
929                             0, HZ*5);
930 
931             dbg("ioctl_scanner(%d): outreq: result:%d\n", scn_minor, result);
932 
933             return result;
934         }
935         default:
936             return -ENOIOCTLCMD;
937         }
938         return 0;
939     }
940     #endif /* SCN_IOCTL */
```

驱动程序的设计者为扫描器的 ioctl( )操作定义了 PV8630_IOCTL_INREQUEST 和 PV8630_IOCTL_OUTREQUEST 两种命令，前者用于从扫描器读入状态信息，后者用于向扫描器发出控制命令。二者都通过 usb_control_msg( )完成信息的传递。比较一下 897 和 925 行，可以看出所不同的有两点。首先是第二个参数，一为 usb_rcvctrlpipe(dev, 0)，一为 usb_sndctrlpipe(dev, 0)；还有就是第 5 个参数中的标志位 USB_DIR_IN 和 USB_DIR_OUT。先看第二个参数，这里引用的两个宏操作均定义于 include/linux/usb.h：

```
700     #define PIPE_CONTROL            2
  ......
745     #define usb_sndctrlpipe(dev,endpoint)   \
                ((PIPE_CONTROL << 30) | __create_pipe(dev,endpoint))
746     #define usb_rcvctrlpipe(dev,endpoint)   \
                ((PIPE_CONTROL << 30) | __create_pipe(dev,endpoint) | USB DIR IN)
```

这里的__create_pipe( )是个 inline 函数，也是在 include/linux/usb.h 中定义的：

```
734     static inline unsigned int __create_pipe(struct usb_device *dev,
                                                unsigned int endpoint)
735     {
736         return (dev->devnum << 8) | (endpoint << 15) | (dev->slow << 26);
737     }
```

这些操作为本次传输构筑起一个 32 位长字，以后将用作传令(token)信包的主体。这个长字的最高

两位表示传输的类型，中部是 7 位的设备地址加上 4 位的端点号。其 bit26 表示对方是否低速设备，bit7 为 1 则表示方向为输入。

常数 USB_DIR_OUT 和 USB_DIR_IN 也定义于 include/linux/usb.h：

```
38    /*
39     * USB directions
40     */
41    #define USB_DIR_OUT          0
42    #define USB_DIR_IN           0x80
```

此外，同一文件中为等时、成块以及中断管道也定义了相应的宏操作以及常数。

```
747   #define usb_sndisocpipe(dev, endpoint)    \
          ((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev, endpoint))
748   #define usb_rcvisocpipe(dev, endpoint)    \
          ((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
749   #define usb_sndbulkpipe(dev, endpoint)    \
          ((PIPE_BULK << 30) | __create_pipe(dev, endpoint))
750   #define usb_rcvbulkpipe(dev, endpoint)    \
          ((PIPE_BULK << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
751   #define usb_sndintpipe(dev, endpoint)     \
          ((PIPE_INTERRUPT << 30) | __create_pipe(dev, endpoint))
752   #define usb_rcvintpipe(dev, endpoint)     \
          ((PIPE_INTERRUPT << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
```

```
698   #define PIPE_ISOCHRONOUS        0
699   #define PIPE_INTERRUPT          1
700   #define PIPE_CONTROL            2
701   #define PIPE_BULK               3
```

再看第 4 个参数，其中的 USB_RECIP_DEVICE 等常数也定义于 include/linux/usb.h 中：

```
21    /*
22     * USB types
23     */
24    #define USB_TYPE_STANDARD       (0x00 << 5)
25    #define USB_TYPE_CLASS          (0x01 << 5)
26    #define USB_TYPE_VENDOR         (0x02 << 5)
27    #define USB_TYPE_RESERVED       (0x03 << 5)
28
29    /*
30     * USB recipients
31     */
32    #define USB_RECIP_MASK          0x1f
33    #define USB_RECIP_DEVICE        0x00
```

```
34      #define USB_RECIP_INTERFACE       0x01
35      #define USB_RECIP_ENDPOINT        0x02
36      #define USB_RECIP_OTHER           0x03
```

常数 USB_RECIP_DEVICE 表示传输的终极对象是设备，而不是接口，也不是端点本身。USB_TYPE_VENDOR 则表示在目标设备中要访问的寄存器是由设备制造商定义的，而不属于标准的 USB 寄存器。作为对比，读者可以回到前一小节看一下枚举阶段对 usb_control_msg( )的调用，在那里第 4 个参数常常是 0 或 USB_DIR_IN，这就是因为 USB_TYPE_STANDARD 和 USB_RECIP_DEVICE 的定义都是 0，USB_DIR_OUT 也是 0。

虽然我们在这里是结合具体的设备扫描器读 usb_control_msg( )的代码，这却是个通用的、属于基础设施一类的函数。在前一小节中，我们就看到在 USB 设备枚举的过程中频繁地调用这个函数从设备读取各种信息。这个函数的代码在 drivers/usb/usb.c 中：

[ioctl_scanner( ) > usb_control_msg( )]

```
1065    /**
1066     * usb_control_msg - Builds a control urb, sends it off and waits for completion
1067     * @dev: pointer to the usb device to send the message to
1068     * @pipe: endpoint "pipe" to send the message to
1069     * @request: USB message request value
1070     * @requesttype: USB message request type value
1071     * @value: USB message value
1072     * @index: USB message index value
1073     * @data: pointer to the data to send
1074     * @size: length in bytes of the data to send
1075     * @timeout: time to wait for the message to complete before
           timing out (if 0 the wait is forever)
1076     *
1077     * This function sends a simple control message to a specified endpoint
1078     * and waits for the message to complete, or timeout.
1079     *
1080     * If successful, it returns 0, othwise a negative error number.
1081     *
1082     * Don't use this function from within an interrupt context, like a
1083     * bottom half handler.  If you need a asyncronous message, or need to send
1084     * a message from within interrupt context, use usb_submit_urb( )
1085     */
1086    int usb_control_msg(struct usb_device *dev, unsigned int pipe,
                    __u8 request, __u8 requesttype,
1087                __u16 value, __u16 index, void *data, __u16 size, int timeout)
1088    {
1089        devrequest *dr = kmalloc(sizeof(devrequest), GFP_KERNEL);
1090        int ret;
1091
1092        if (!dr)
1093            return -ENOMEM;
```

```
1094
1095        dr->requesttype = requesttype;
1096        dr->request = request;
1097        dr->value = cpu_to_le16p(&value);
1098        dr->index = cpu_to_le16p(&index);
1099        dr->length = cpu_to_le16p(&size);
1100
1101        //dbg("usb_control_msg");
1102
1103        ret = usb_internal_control_msg(dev, pipe, dr, data, size, timeout);
1104
1105        kfree(dr);
1106
1107        return ret;
1108    }
```

参数 dev 指向目标设备的 usb_device 数据结构。pipe 是个 32 位无符号整数，那就是上面构筑的 32 位长字。其最高两位表示传输的类型(等时/中断/控制/成块)，其余各位包括对方的端点号以及设备号，以及设备是否为全速(或低速)。此外，requesttype 是个 8 位字节，其最高位表示传输的方向，最低 5 位则表明传输终极对象的类别(设备/接口/端点/其它)，通常这意味着一组寄存器或者一小块存储区间；而 index 则进而指明了具体的单元，这就是终极的操作对象。针对这个操作对象，request 说明了需要进行的操作，而 value 则是参数。如果有更多的数据需要传递(读/写)，则通过缓冲区 data 进行，其大小为 size。这些都是从用户空间传下的参数，而传输的目的正是要把这些信息发送给目标设备。最后，参数 timeout 表示愿意睡眠等待传输完成的时间。显然，usb_control_msg( )只是个外包装，实际的操作由 usb_internal_control_msg( )完成。在传输的过程中，端点起着类似于“传达室”的作用。

根据 USB 总线规格书的规定，需要发送给目标设备的这些信息要组装在一个操作请求块中。这里的 devrequest 就是根据这个规定而定义的数据结构，其定义在 include/linux/usb.h 中：

```
151    typedef struct {
152           __u8 requesttype;
153           __u8 request;
154           __u16 value;
155           __u16 index;
156           __u16 length;
157    } devrequest __attribute__ ((packed));
```

实际上，这个数据结构就是 SETUP 信包的内容，而缓冲区的内容，则就是随后的数据信包的内容。至于状态阶段的信包，则是来自(或去向) 终极操作对象(而不是交互对象)的确认。

函数 usb_internal_control_msg( )的代码在 drivers/usb/usb.c 中：

[ioctl_scanner( ) > usb_control_msg( ) > usb_internal_control_msg( )]

```
1042    // returns status (negative) or length (positive)
1043    int usb_internal_control_msg(struct usb_device *usb_dev, unsigned int pipe,
```

```
1044                    devrequest *cmd,  void *data, int len, int timeout)
1045    {
1046        urb_t *urb;
1047        int retv;
1048        int length;
1049
1050        urb = usb_alloc_urb(0);
1051        if (!urb)
1052            return -ENOMEM;
1053
1054        FILL_CONTROL_URB(urb, usb_dev, pipe, (unsigned char*)cmd,
                        data, len,     /* build urb */
1055                (usb_complete_t)usb_api_blocking_completion,0);
1056
1057        retv = usb_start_wait_urb(urb, timeout, &length);
1058        if (retv < 0)
1059            return retv;
1060        else
1061            return length;
1062
1063    }
```

对于 USB 总线上的每个传输，需要为之创建一个“USB 传输请求块”，即 usb 数据结构。简而言之，发送控制报文的过程就是：根据参数建立一个 usb 数据结构，把这个 usb 结构交给低层，让低层据以调度相应的控制传输，然后睡眠等待传输的完成。

这种数据结构的定义在 include/linux/usb.h 中：

```
440     typedef struct urb
441     {
442         spinlock_t lock;              /* lock for the URB */
443         void *hcpriv;                 /* private data for host controller */
444         struct list_head urb_list;    /* list pointer to all active urbs  */
445         struct urb *next;             /* pointer to next URB  */
446         struct usb_device *dev;       /* pointer to associated USB device */
447         unsigned int pipe;            /* pipe information */
448         int status;                   /* returned status */
449         unsigned int transfer_flags;  /* USB_DISABLE_SPD | USB_ISO_ASAP | etc. */
450         void *transfer_buffer;        /* associated data buffer */
451         int transfer_buffer_length;   /* data buffer length */
452         int actual_length;            /* actual data buffer length */
453         int bandwidth;        /* bandwidth for this transfer request (INT or ISO) */
454         unsigned char *setup_packet;      /* setup packet (control only) */
455
456         int start_frame;              /* start frame (iso/irq only) */
457         int number_of_packets;        /* number of packets in this request (iso) */
458         int interval;                 /* polling interval (irq only) */
459         int error_count;              /* number of errors in this transfer (iso only) */
```

```
460        int timeout;                /* timeout (in jiffies) */
461
462        void *context;              /* context for completion routine */
463        usb_complete_t complete;    /* pointer to completion routine */
464
465        iso_packet_descriptor_t iso_frame_desc[0];
466    } urb_t, *purb_t;
```

同一文件中还定义了一个宏操作 FILL_CONTROL_URB( ):

```
468    #define FILL_CONTROL_URB(a, aa, b, c, d, e, f, g)  \
469        do {\
470        spin_lock_init(&(a)->lock);\
471        (a)->dev=aa;\
472        (a)->pipe=b;\
473        (a)->setup_packet=c;\
474        (a)->transfer_buffer=d;\
475        (a)->transfer_buffer_length=e;\
476        (a)->complete=f;\
477        (a)->context=g;\
478        } while (0)
```

对照上面对 FILL_CONTROL_URB( )的引用，可知 urb->dev 设置成指向 usb_dev，urb->pipe 为目标设备的控制管道，即端点 0，urb->setup_packet 指向前面创建的 devrequest 数据结构，urb->transfer_buffer 设置成前面的参数 data，urb->complete 指向 usb_api_blocking_completion( )，urb->context 则暂时为 0。

设置好 urb 数据结构以后，就通过 usb_start_wait_urb( )“提交”这个 urb 数据结构，并等待交互的完成。这个函数的代码在 drivers\usb\usb.c 中：

[ioctl_scanner( ) > usb_control_msg( ) > usb_internal_control_msg( ) > usb_start_wait_urb( )]

```
995    // Starts urb and waits for completion or timeout
996    static int usb_start_wait_urb(urb_t *urb, int timeout, int* actual_length)
997    {
998        DECLARE_WAITQUEUE(wait, current);
999        DECLARE_WAIT_QUEUE_HEAD(wqh);
1000       api_wrapper_data awd;
1001       int status;
1002
1003       awd.wakeup = &wqh;
1004       init_waitqueue_head(&wqh);
1005       current->state = TASK_INTERRUPTIBLE;
1006       add_wait_queue(&wqh, &wait);
1007       urb->context = &awd;
1008       status = usb_submit_urb(urb);
1009       if (status) {
```

```
1010            // something went wrong
1011            usb_free_urb(urb);
1012            current->state = TASK_RUNNING;
1013            remove_wait_queue(&wqh, &wait);
1014            return status;
1015        }
1016
1017        if (urb->status == -EINPROGRESS) {
1018            while (timeout && urb->status == -EINPROGRESS)
1019                status = timeout = schedule_timeout(timeout);
1020        } else
1021            status = 1;
1022
1023        current->state = TASK_RUNNING;
1024        remove_wait_queue(&wqh, &wait);
1025
1026        if (!status) {
1027            // timeout
1028            printk("usb control/bulk_msg: timeout\n");
1029            usb_unlink_urb(urb);  // remove urb safely
1030            status = -ETIMEDOUT;
1031        } else
1032            status = urb->status;
1033
1034        if (actual_length)
1035            *actual_length = urb->actual_length;
1036
1037        usb_free_urb(urb);
1038        return status;
1039    }
```

像以前多次见到的那样(详见第 4 章)，在当前进程的系统空间堆栈上建立起一个等待队列，并将当前进程的 task 结构通过一个 wait_queue_t 数据结构挂在队列中。同时，又定义了一个 api_wrapper_data 数据结构 awd。这种数据结构定义于 include/linux/usb.h：

```
540    typedef struct
541    {
542      wait_queue_head_t *wakeup;
543
544      void* stuff;
545      /* more to follow */
546    } api_wrapper_data;
```

使 awd.wakeup 指向上述的队列头，再使 urb 结构中的指针 context 指向 awd，就在 urb 结构与等待队列之间建立起了联系。这样，从具体的 urb 结构出发，就可以找到正在等待其(所代表的传输)完成的进程。然后就通过 usb_submit_urb( )提交这个传输请求。其代码在 drivers/usb/usb.c 中：

[ioctl_scanner( ) > usb_control_msg( ) > usb_internal_control_msg( ) > usb_start_wait_urb( ) > usb_submit_urb( )]

```
955    int usb_submit_urb(urb_t *urb)
956    {
957        if (urb && urb->dev)
958            return urb->dev->bus->op->submit_urb(urb);
959        else
960            return -ENODEV;
961    }
```

这里使用了“提交”(submit)这个词，但并不是向上层提交，相反倒是交付给下层的意思。那么下层是什么呢？这就是 UHCI 或 OHCI。对于采用 UHCI 界面的 USB 总线控制器，其 usb_bus 结构中的 usb_operations 结构指针 op 指向 drivers/usb/uhci.c 中定义的 uhci_device_operations。

```
1615   struct usb_operations uhci_device_operations = {
1616       uhci_alloc_dev,
1617       uhci_free_dev,
1618       uhci_get_current_frame_number,
1619       uhci_submit_urb,
1620       uhci_unlink_urb
1621   };
```

显然，UHCI 界面的函数指针 submit_urb 指向 uhci_submit_urb( )。这是个通用的函数，不但用来交付控制传输，也用来交付等时、中断以及成块传输。其代码在 drivers/usb/uhci.c 中：

[ioctl_scanner( ) > usb_control_msg( ) > usb_internal_control_msg( ) > usb_start_wait_urb( ) > usb_submit_urb( ) > uhci_submit_urb( )]

```
1291   static int uhci_submit_urb(struct urb *urb)
1292   {
1293       int ret = -EINVAL;
1294       struct uhci *uhci;
1295       unsigned long flags;
1296       struct urb *u;
1297       int bustime;
1298
1299       if (!urb)
1300           return -EINVAL;
1301
1302       if (!urb->dev || !urb->dev->bus || !urb->dev->bus->hcpriv)
1303           return -ENODEV;
1304
1305       uhci = (struct uhci *)urb->dev->bus->hcpriv;
1306
1307       /* Short circuit the virtual root hub */
```

```
1308        if (usb_pipedevice(urb->pipe) == uhci->rh.devnum)
1309            return rh_submit_urb(urb);
1310
1311        u = uhci_find_urb_ep(uhci, urb);
1312        if (u && !(urb->transfer_flags & USB_QUEUE_BULK))
1313            return -ENXIO;
1314
1315        usb_inc_dev_use(urb->dev);
1316        spin_lock_irqsave(&urb->lock, flags);
1317
1318        if (!uhci_alloc_urb_priv(urb)) {
1319            spin_unlock_irqrestore(&urb->lock, flags);
1320            usb_dec_dev_use(urb->dev);
1321
1322            return -ENOMEM;
1323        }
1324
1325        switch (usb_pipetype(urb->pipe)) {
1326        case PIPE_CONTROL:
1327            ret = uhci_submit_control(urb);
1328            break;
1329        case PIPE_INTERRUPT:
1330            if (urb->bandwidth == 0) {  /* not yet checked/allocated */
1331                bustime = usb_check_bandwidth(urb->dev, urb);
1332                if (bustime < 0)
1333                    ret = bustime;
1334                else {
1335                    ret = uhci_submit_interrupt(urb);
1336                    if (ret == -EINPROGRESS)
1337                        usb_claim_bandwidth(urb->dev, urb, bustime, 0);
1338                }
1339            } else      /* bandwidth is already set */
1340                ret = uhci_submit_interrupt(urb);
1341            break;
1342        case PIPE_BULK:
1343            ret = uhci_submit_bulk(urb, u);
1344            break;
1345        case PIPE_ISOCHRONOUS:
1346            if (urb->bandwidth == 0) {  /* not yet checked/allocated */
1347                if (urb->number_of_packets <= 0) {
1348                    ret = -EINVAL;
1349                    break;
1350                }
1351                bustime = usb_check_bandwidth(urb->dev, urb);
1352                if (bustime < 0) {
1353                    ret = bustime;
1354                    break;
1355                }
```

```
1356
1357                ret = uhci_submit_isochronous(urb);
1358                if (ret == -EINPROGRESS)
1359                    usb_claim_bandwidth(urb->dev, urb, bustime, 1);
1360            } else      /* bandwidth is already set */
1361                ret = uhci_submit_isochronous(urb);
1362            break;
1363        }
1364
1365        urb->status = ret;
1366
1367        spin_unlock_irqrestore(&urb->lock, flags);
1368
1369        if (ret == -EINPROGRESS)
1370            ret = 0;
1371        else {
1372            uhci_unlink_generic(urb);
1373            usb_dec_dev_use(urb->dev);
1374        }
1375
1376        return ret;
1377    }
```

这是个需要细加阅读的函数。

对于采用 UHCI 界面的 USB 总线控制器，其 usb_bus 结构中的指针 hcpriv 指向一个 uhci 数据结构，而 uhci 结构中有个次层结构 rh，里面是有关根集中器的信息，这是在根集中器初始化时设置好的。如果目标设备恰好就是根集中器，那么通信的过程可以简化，因为 CPU 直接就可以访问其各个寄存器，不需要通过 USB 总线就能进行通信，所以此时由 rh_submit_urb( )完成操作。这个函数的代码在 drivers/usb/uhci.c 中，但是我们在这里就从略了。

与其他所有 USB 设备的通信(传输)都要通过 USB 总线上的交互来完成，因而需要为具体的传输调度一个或几个交互。除等时传输之外，在调度中不允许同时存在对同一对象的两个同种传输。这是为什么呢？我们以前讲过，控制传输和成块传输是以传输（而不是交互）为单位挂入调度队列的，一个传输就是一个交互队列，而对这些队列的执行又可以是横向执行。如果对同一对象的两个传输都挂在调度队列中，就有可能使两个传输中的交互夹杂在一起；例如从第一个传输中执行了一个交互以后就在第二个传输中也执行一个交互，然后又回到第一个传输，这当然就乱了套。至于中断传输，则只包含一个交互，并且是以交互为单位进行调度的，但是却并没有理由对同一端点调度两个中断传输。

所以，先通过 uhci_find_urb_ep( )，寻找已经调度而尚未完成的对同一对象的同种传输，其代码在 drivers/usb/uhci.c 中：

[ioctl_scanner( ) > usb_control_msg( ) > usb_internal_control_msg( ) > usb_start_wait_urb( ) > usb_submit_urb( ) > uhci_submit_urb( ) > uhci_find_urb_ep( )]

```
1263    static struct urb *uhci_find_urb_ep(struct uhci *uhci, struct urb *urb)
1264    {
```

```
1265        struct list_head *tmp, *head = &uhci->urb_list;
1266        unsigned long flags;
1267        struct urb *u = NULL;
1268
1269        if (usb_pipeisoc(urb->pipe))
1270            return NULL;
1271
1272        nested_lock(&uhci->urblist_lock, flags);
1273        tmp = head->next;
1274        while (tmp != head) {
1275            u = list_entry(tmp, struct urb, urb_list);
1276
1277            tmp = tmp->next;
1278
1279            if (u->dev == urb->dev &&
1280                u->pipe == urb->pipe)
1281                goto found;
1282        }
1283        u = NULL;
1284
1285    found:
1286        nested_unlock(&uhci->urblist_lock, flags);
1287
1288        return u;
1289    }
```

等时交互的调度是个不同的问题，所以如果对方是个等时端点（1269 行）就立即返回 0。这里的 usb_pipeisoc( )是个宏操作，定义于 include/linux/usb.h。与此类似的宏操作还有 usb_pipeint( )、usb_pipecontrol( )以及 usb_pipebulk( )。

```
716    #define usb_pipeisoc(pipe)   (usb_pipetype((pipe)) == PIPE_ISOCHRONOUS)
717    #define usb_pipeint(pipe)    (usb_pipetype((pipe)) == PIPE_INTERRUPT)
718    #define usb_pipecontrol(pipe)    (usb_pipetype((pipe)) == PIPE_CONTROL)
719    #define usb_pipebulk(pipe)   (usb_pipetype((pipe)) == PIPE_BULK)
```

代码中的 nested_lock( )和 nested_unlock 也是 drivers/usb/uhci.h 中定义的宏操作：

```
21    #define nested_lock(snl, flags) \
22        if ((snl)->uniq == current) { \
23            (snl)->count++; \
24            flags = 0; /* No warnings */ \
25        } else { \
26            spin_lock_irqsave(&(snl)->lock, flags); \
27            (snl)->count++; \
28            (snl)->uniq = current; \
29        }
30
```

```
31    #define nested_unlock(snl, flags) \
32        if (!--(snl)->count) { \
33            (snl)->uniq = NULL; \
34            spin_unlock_irqrestore(&(snl)->lock, flags); \
35        }
```

如果在队列 urb_list 中找到了目标设备相同，而且端点也相同的 urb 结构，就返回指向该结构的指针，否则返回 NULL。一般而言，如果已经存在对同一目标的同种传输，就应该拒绝调度新的传输请求，让高层的软件决定是否稍后再来试试。其实，控制传输都是很短暂的，并且启动控制传输的进程要睡眠等待其完成，所以发生这种情况的可能性本来就很小。但是，对于成块传输则情况有所不同，所以允许将新的传输与已经存在的传输合并，但是必须将对同一对象的传输“串行化”，就是把它们的交互依次连接在同一个队列中。这样，不管是横向执行还是纵向执行都能保证正确的次序。

除 urb 数据结构以外，还需要有个 urb_priv 数据结构与其配合使用。这种数据结构定义于 drivers/usb/uhci.h：

```
337    struct urb_priv {
338        struct urb *urb;
339
340        struct uhci_qh *qh;      /* QH for this URB */
341
342        int fsbr : 1;            /* URB turned on FSBR */
343        int fsbr_timeout : 1;        /* URB timed out on FSBR */
344        int queued : 1;          /* QH was queued (not linked in) */
345        int short_control_packet : 1;   /* If we get a short packet during */
346                        /*  a control transfer, retrigger */
347                        /*  the status phase */
348
349        unsigned long inserttime;   /* In jiffies */
350
351        struct list_head list;
352
353        struct list_head urb_queue_list;    /* URB's linked together */
354    };
```

函数 uhci_alloc_urb_priv( )的代码在 drivers/usb/uhci.c 中：

[usb_control_msg( ) > usb_internal_control_msg( ) > usb_start_wait_urb( ) > usb_submit_urb( ) > uhci_submit_urb( ) > uhci_alloc_urb_priv( )]

```
480    struct urb_priv *uhci_alloc_urb_priv(struct urb *urb)
481    {
482        struct urb_priv *urbp;
483
484        urbp = kmem_cache_alloc(uhci_up_cachep, in_interrupt( ) ?
                                        SLAB_ATOMIC : SLAB_KERNEL);
```

```
485         if (!urbp)
486             return NULL;
487 
488         memset((void *)urbp, 0, sizeof(*urbp));
489 
490         urbp->inserttime = jiffies;
491         urbp->urb = urb;
492 
493         INIT_LIST_HEAD(&urbp->list);
494         INIT_LIST_HEAD(&urbp->urb_queue_list);
495 
496         urb->hcpriv = urbp;
497 
498         return urbp;
499     }
```

下面的操作，就取决于传输的类型，也就是目标端点的类型（1325 行）了。我们先结合 usb_control_msg( )的情景看控制传输的调度，以后还要再回过来看其他几种传输。

控制传输是通过 uhci_submit_control( )调度和实现的，其代码在 drivers/usb/uhci.c 中：

[ioctl_scanner( ) > usb_control_msg( ) > usb_internal_control_msg( ) > usb_start_wait_urb( ) > usb_submit_urb( ) > uhci_submit_urb( ) > uhci_submit_control( )]

```
633     /*
634      * Control transfers
635      */
636     static int uhci_submit_control(struct urb *urb)
637     {
638         struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;
639         struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;
640         struct uhci_td *td;
641         struct uhci_qh *qh;
642         unsigned long destination, status;
643         int maxsze = usb_maxpacket(urb->dev, urb->pipe, usb_pipeout(urb->pipe));
644         int len = urb->transfer_buffer_length;
645         unsigned char *data = urb->transfer_buffer;
646 
647         /* The "pipe" thing contains the destination in bits 8--18 */
648         destination = (urb->pipe & PIPE_DEVEP_MASK) | USB_PID_SETUP;
649 
650         /* 3 errors */
651         status = (urb->pipe & TD_CTRL_LS) | TD_CTRL_ACTIVE | (3 << 27);
652 
653         /*
654          * Build the TD for the control request
655          */
656         td = uhci_alloc_td(urb->dev);
```

```
657         if (!td)
658             return -ENOMEM;
659 
660         uhci_add_td_to_urb(urb, td);
661         uhci_fill_td(td, status, destination | (7 << 21),
662             virt_to_bus(urb->setup packet));
663 
664         /*
665          * If direction is "send", change the frame from SETUP (0x2D)
666          * to OUT (0xE1). Else change it from SETUP to IN (0x69).
667          */
668         destination ^= (USB_PID_SETUP ^ usb_packetid(urb->pipe));
669 
670         if (!(urb->transfer_flags & USB_DISABLE_SPD))
671             status |= TD_CTRL_SPD;
672 
673         /*
674          * Build the DATA TD's
675          */
676         while (len > 0) {
677             int pktsze = len;
678 
679             if (pktsze > maxsze)
680                 pktsze = maxsze;
681 
682             td = uhci_alloc_td(urb->dev);
683             if (!td)
684                 return -ENOMEM;
685 
686             /* Alternate Data0/1 (start with Data1) */
687             destination ^= 1 << TD_TOKEN_TOGGLE;
688 
689             uhci_add_td_to_urb(urb, td);
690             uhci_fill_td(td, status, destination | ((pktsze - 1) << 21),
691                 virt_to_bus(data));
692 
693             data += pktsze;
694             len -= pktsze;
695         }
696 
697         /*
698          * Build the final TD for control status
699          */
700         td = uhci_alloc_td(urb->dev);
701         if (!td)
702             return -ENOMEM;
703 
704         /*
```

```
705          * It's IN if the pipe is an output pipe or we're not expecting
706          * data back.
707          */
708         destination &= ~TD_PID;
709         if (usb_pipeout(urb->pipe) || !urb->transfer_buffer_length)
710             destination |= USB_PID_IN;
711         else
712             destination |= USB_PID_OUT;
713 
714         destination |= 1 << TD_TOKEN_TOGGLE;        /* End in Data1 */
715 
716         status &= ~TD_CTRL_SPD;
717 
718         uhci_add_td_to_urb(urb, td);
719         uhci_fill_td(td, status | TD_CTRL_IOC,
720             destination | (UHCI_NULL_DATA_SIZE << 21), 0);
721 
722         qh = uhci_alloc_qh(urb->dev);
723         if (!qh)
724             return -ENOMEM;
725 
726         /* Low speed or small transfers gets a different queue and treatment */
727         if (urb->pipe & TD_CTRL_LS) {
728             uhci_insert_tds_in_qh(qh, urb, 0);
729             uhci_insert_qh(uhci, &uhci->skel_ls_control_qh, qh);
730         } else {
731             uhci_insert_tds_in_qh(qh, urb, 1);
732             uhci_insert_qh(uhci, &uhci->skel_hs_control_qh, qh);
733             uhci_inc_fsbr(uhci, urb);
734         }
735 
736         urbp->qh = qh;
737 
738         uhci_add_urb_list(uhci, urb);
739 
740         return -EINPROGRESS;
741     }
```

一次控制传输至少要由两个交互完成。第一个交互是由主机通过 USB 总线控制器向目标设备的控制端点发送一个 SETUP 报文，这就是所谓的 SETUP 阶段。然后，根据需要传输的数据量大小而有一个或几个数据交互，每个交互传递一个 DATA 报文，传递的方向既可以是输出也可以是输入，这就是数据阶段。不过，如果没有数据就不需要传送 DATA 报文。最后，还要有一个状态交互，让 DATA 报文的接收者向对方发送一个状态报文，确认对 SETUP 报文或 DATA 报文的接收，这就是状态阶段。这样，最少是两个交互，通常则是三个交互。交互是 USB 控制器的执行单位。每个交互都由一个 uhci_td 数据结构代表，我们已经在前面看到过 uhci_td 数据结构的定义。如前所述，uhci_td 数据结构中的前 4 个 32 位长字是为 USB 控制器的硬件准备的，实际上相当于 4 个寄存器，作用分别如下：

(1) link。相当于“链接寄存器”。其高 28 位后面添上 4 位 0 成为一个指针，指向下一个 uhci_td 结构或 uhci_qh 结构的物理地址。其低 4 位则为标志位，其中 bit0 为 1 表示链接的终结，bit1 为 1 表示指针所指为队列头，即 uhci_qh 结构，否则为 uhci_td 结构。至于 bit2，则为 1 时表示纵向执行，为 0 时表示横向执行。

(2) status。相当于“控制/状态寄存器”。其中的 TD_CTRL_ACTIVE 状态位起着特殊的作用，为 1 时表示等待执行，为 0 时则表示已经执行。此外，寄存器中还有个重发次数位段，表示在出错的情况下允许重发几次。代码中（651 行）把重发次数设置成 3。 USB 控制器在每一次出错重发时都把这个计数减 1，如果达到了 0 就认为彻底失败了。

(3) info。相当于“命令寄存器”，其内容包括传输的类型及目标，实际上就是 token 信包的主体。

(4) buffer。相当于“缓冲区地址寄存器”，指向发送/接收数据的缓冲区。

每个信包的头部都包含着一个 8 位的“信包标识码”PID，表明具体信包的类型。SETUP 信包的 PID 就是 USB_PID_SETUP，定义于 include/linux/usb.h：

```
81     /*
82      * USB Packet IDs (PIDs)
83      */
84     #define USB_PID_UNDEF_0              0xf0
85     #define USB_PID_OUT                  0xe1
86     #define USB_PID_ACK                  0xd2
87     #define USB_PID_DATA0                0xc3
88     #define USB_PID_PING                 0xb4 /* USB 2.0 */
89     #define USB_PID_SOF                  0xa5
90     #define USB_PID_NYET                 0x96 /* USB 2.0 */
91     #define USB_PID_DATA2                0x87 /* USB 2.0 */
92     #define USB_PID_SPLIT                0x78 /* USB 2.0 */
93     #define USB_PID_IN                   0x69
94     #define USB_PID_NAK                  0x5a
95     #define USB_PID_DATA1                0x4b
96     #define USB_PID_PREAMBLE             0x3c /* Token mode */
97     #define USB_PID_ERR                  0x3c /* USB 2.0: handshake mode */
98     #define USB_PID_SETUP                0x2d
99     #define USB_PID_STALL                0x1e
100    #define USB_PID_MDATA                0x0f /* USB 2.0 */
```

这些定义来自 USB 的规格书，其中有些是为 USB 2.0 定义的。

代码中先为 SETUP 交互分配一个 uhci_td 数据结构（656 行），同时还要为报文准备下 destination 和 status 两个 32 位字段。前者用于“命令寄存器”，其内容就是 token 信包的主体，包含着交互对象的地址和端点号，还有 SETUP 信包的 PID，即 USB_PID_SETUP。此外，信包的长度也组装在这个长字中，SETUP 信包的实际长度为 8 个字节（7＋1）。

对于 SETUP 交互，要发送的报文是对目标设备的操作命令，这就是前面准备好的 devrequest 数据结构，urb->setup_packet 就指向这个结构。

如上所述，“控制/状态寄存器”中的 TD_CTRL_ACTIVE 状态位起着特殊的作用，驱动软件在调度一个交互请求时将这一位设成 1，表示这是个待执行的“活跃”交互；而 USB 控制器，则在完成了

该次交互以后（或成功，或彻底失败）就将这一位改成 0。这样，驱动软件只要扫描各个 uhci_td 数据结构，发现某个 uhci_td 数据结构的 TD_CTRL_ACTIVE 位变成了 0，就说明这个交互已经完成。

分配了一个 uhci_td 数据结构以后，就通过 uhci_add_td_to_urb( )把它链入 urb_priv 结构中的交互描述块队列（660 行），并通过 inline 函数 uhci_fill_td( )设置 uhci_td 结构中的几个重要字段（见前）。这样，就相当于设置好了 USB 控制器的 4 个寄存器。数据信包的内容来自缓冲区 urb->setup_packet，交互描述块中的字段 buffer 设置成缓冲区的物理地址；在发送数据信包时，USB 控制器就通过 DMA 从这个缓冲区中读取数据。准备好了一个交互请求，就通过 uhci_add_td_to_urb( )，将其挂入传输请求即 urb 结构内的队列中（实际上是在 urb_priv 结构内）。这个函数的代码在 drivers/usb/uhci.c 中：

[ioctl_scanner( ) > usb_control_msg( ) > usb_internal_control_msg( ) > usb_start_wait_urb( ) > usb_submit_urb( ) > uhci_submit_urb( ) > uhci_submit_control( ) > uhci_add_td_to_urb( )]

```
501	static void uhci_add_td_to_urb(struct urb *urb, struct uhci_td *td)
502	{
503		struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;
504	
505		td->urb = urb;
506	
507		list_add_tail(&td->list, &urbp->list);
508	}
```

至此，已经为控制传输的第一阶段，即 SETUP 阶段准备好了需要传输的报文。USB 控制器在执行时将首先通过 token 信包在 USB 总线上发布一个通告，说明传输的类型(控制)、交互的对象(扫描器的控制端点)、信包的种类(SETUP)。然后，就会把前面准备好的 devrequest 数据结构作为 SETUP 信包，即第一阶段的数据发送出去。这个信包的内容告诉扫描器想要干什么，例如对扫描器内的哪一个寄存器进行何种操作。由于具体的寄存器是由扫描器的制造商定义的，这些内容对于 USB 总线是透明的。而扫描器，则在接收到这个 SETUP 信包以后要发回一个 ACK 信包加以确认。

如果所要求的操作需要有附加的数据传递，使有 DATA 报文要传送，但是其类型不再是 USB_PID_SETUP，而是 USB_PID_IN 或 USB_PID_OUT 了。所以一方面把 destination 中的 USB_PID_SETUP 去除，另一方面根据所用的管道确定是 USB_PID_IN 还是 USB_PID_OUT。这里的宏操作 usb_packetid( )定义于 include/linux/usb.h：

```
706	#define usb_packetid(pipe)	(((pipe) & USB_DIR_IN) ? USB_PID_IN : USB_PID_OUT)
```

在 SETUP 报文以后有几个 DATA 信包取决于数据量和允许的信包大小。当有多个 DATA 信包需要传送时，需要给每个信包编上序号，使接收方能知道是否丢失了信包。为此目的，在 DATA 信包的头部采用了一个二进制位作为序号，其位置为 TD_TOKEN_TOGGLE，每（调度）发送一个信包就将这一位翻转一次(687)，使序号为 0 与为 1 的 DATA 信包相间。因此，DATA 信包又可以分成 DATA0 和 DATA1 两种。如果接收方连续收到两个序号相同的信包，就知道一定是丢失了一个信包。当然，这是建立在不会接连丢失两个信包这么个前提下的。

最后是一次“状态”(status)交互，由接收方发送一个确认报文。所以，如果整个传输(数据信包)

是输出，那么确认信包就应该由设备向主机发送，所以对主机而言是 USB_PID_IN。或者，如果根本就没有数据信包，那也应该是 USB_PID_IN；否则就是 USB_PID_OUT。此外，确认信包的序号总是 1（714 行），没有数据部分（720 行），也不需要检测信包的长度。由于状态交互是控制传输中的最后一个交互，这里(719 行)还把标志位 TD_CTRL_IOC 设成 1，表示这个交互完成后要向 CPU 发出一个中断请求。注意不要将状态交互与握手信包相混淆，状态交互同样包括传令、数据和握手三个信包的传递。

这样，一个传输的所有交互描述块就都挂入了 urb 数据结构内的队列中。可是，这并不是最终的目的，最终的目的是要挂入 USB 总线的调度队列中。可是，这里有个问题。USB 总线的调度队列是供 USB 控制器硬件在“执行”时使用的，链接指针必须采用物理地址。可是，采用了物理地址，CPU 就不能顺着链接指针依次访问队列中的各个成分了。反过来，如果要考虑 CPU 的访问以及队列操作，就要采用虚拟地址；而若采用了虚拟地址，USB 控制器硬件就不能顺着链接依次执行了。对于具有“智能 DMA”功能的设备，这是个共同的问题。解决的办法是让采用虚拟地址和物理地址的链接手段在数据结构中同时并存。所以，在 uhci_td 数据结构的设计中就考虑到了这两方面的需要。里面既有常规的(虚拟地址)队列头 list，让 CPU 可以对 uhci_td 数据结构进行常规的队列操作；又有硬件所需的物理地址指针 link。前面的 uhci_add_td_to_urb( )已经将所有的 uhci_td 数据结构通过它们的队列头 list 连成了常规的队列，下面就要再将这些 uhci_td 数据结构通过物理地址指针 link 链接起来。另一方面，控制传输是作为一个整体，即以 uhci_qh 数据结构为队列头的交互描述块队列挂入调度系统的，所以先要通过 uhci_alloc_qh( )为之分配一个 uhci_qh 数据结构作为队列头部。其代码见 drivers/usb/uhci.c。

[usb_control_msg( ) > usb_internal_control_msg( ) > usb_start_wait_urb( ) > usb_submit_urb( ) > uhci_submit_urb( ) > uhci_submit_control( ) > uhci_alloc_qh( )]

```
302    static struct uhci_qh *uhci_alloc_qh(struct usb_device *dev)
303    {
304        struct uhci_qh *qh;
305
306        qh = kmem_cache_alloc(uhci_qh_cachep, in_interrupt( ) ?
                                              SLAB_ATOMIC : SLAB_KERNEL);
307        if (!qh)
308            return NULL;
309
310        qh->element = UHCI_PTR_TERM;
311        qh->link = UHCI_PTR_TERM;
312
313        qh->dev = dev;
314        qh->prevqh = qh->nextqh = NULL;
315
316        INIT_LIST_HEAD(&qh->remove_list);
317
318        usb_inc_dev_use(dev);
319
320        return qh;
321    }
```

然后，就要看对方是低速设备还是全速设备了，因为二者在 uhci 数据结构中有不同的控制交互队列。不过，二者的操作基本上是相同的，我们在这里只是看全速设备的情景。首先，通过 uhci_insert_tds_in_qh( )将所有的 uhci_td 数据结构通过物理地址链接成一个队列，而上面分配的 uhci_qh 数据结构就是队列的头。这个函数的代码在 drivers/usb/uhci.c 中：

[usb_control_msg( ) > usb_internal_control_msg( ) > usb_start_wait_urb( ) > usb_submit_urb( ) > uhci_submit_urb( ) > uhci_submit_control( ) > uhci_insert_tds_in_qh( )]

```
252    /*
253     * Inserts a td into qh list at the top.
254     */
255    static void uhci_insert_tds_in_qh(struct uhci_qh *qh, struct urb *urb,
                                                               int breadth)
256    {
257        struct list_head *tmp, *head;
258        struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;
259        struct uhci_td *td, *prevtd;
260
261        if (!urbp)
262            return;
263
264        head = &urbp->list;
265        tmp = head->next;
266        if (head == tmp)
267            return;
268
269        td = list_entry(tmp, struct uhci_td, list);
270
271        /* Add the first TD to the QH element pointer */
272        qh->element = virt_to_bus(td) | (breadth ? 0 : UHCI_PTR_DEPTH);
273
274        prevtd = td;
275
276        /* Then link the rest of the TD's */
277        tmp = tmp->next;
278        while (tmp != head) {
279            td = list_entry(tmp, struct uhci_td, list);
280
281            tmp = tmp->next;
282
283            prevtd->link = virt_to_bus(td) | (breadth ? 0 : UHCI_PTR_DEPTH);
284
285            prevtd = td;
286        }
287
288        prevtd->link = UHCI_PTR_TERM;
289    }
```

在 uhci_qh 结构中有两个采用物理地址的指针。一个是 link，用于队列之间的链接，称为横向链接；另一个则是 element，指向本队列中的第一个 uhci_td 结构，这是纵向链接。从代码中可见，首先将队列头的指针 element 设置成第一个 uhci_td 结构的物理地址，然后就依次使各个 uhci_td 结构中的指针 link 设置成下一个 uhci_td 结构的物理地址。不过，无论是 uhci_qh 结构还是 uhci_td 结构的地址都是与 16 字节边界对齐的，所以这些指针中的低 4 位都可以用作控制位。drivers/usb/usb-uhci.h 中定义了这些控制位：

```
72      #define UHCI_PTR_BITS          0x000F
73      #define UHCI_PTR_TERM          0x0001
74      #define UHCI_PTR_QH            0x0002
75      #define UHCI_PTR_DEPTH         0x0004
```

其中之一就是 UHCI_PTR_DEPTH，定义为 4，即 bit2。当这个控制位为 1 时，表示 USB 控制器在执行时应该深度优先，即纵向执行；否则即为宽度优先，即横向执行。所以，对于低速设备因参数 breadth 为 0 而纵向执行；对全速设备则参数 breadth 为 1 而横向执行。此外，队列中最后一个 uhci_td 结构的指针 link 设置成 UHCI_PTR_TERM，就是地址为 0，而最低位(称为“T”位)为 1，表示队列已经到了末尾。还有一位，即 UHCI_PTR_QH，则表示所指向的是 uhci_qh 结构还是 uhci_td 结构。

我们已经建立起了一个以 uhci_qh 结构为头、通过物理地址链接的 uhci_td 结构队列，但这只是个游离于 USB 总线调度系统以外的队列，下一步要通过 uhci_insert_qh( )把这个队列插入 USB 总线调度系统的队列。对于全速设备就是插在 uhci->skel_hs_control_qh 中，其代码见 drivers/usb/uhci.c：

[usb_control_msg( ) > usb_internal_control_msg( ) > usb_start_wait_urb( ) > usb_submit_urb( ) > uhci_submit_urb( ) > uhci_submit_control( ) > uhci_insert_qh( )]

```
331     static void uhci_insert_qh(struct uhci *uhci, struct uhci_qh *skelqh,
                                                        struct uhci_qh *qh)
332     {
333         unsigned long flags;
334
335         spin_lock_irqsave(&uhci->framelist_lock, flags);
336
337         /* Fix the linked list pointers */
338         qh->nextqh = skelqh->nextqh;
339         qh->prevqh = skelqh;
340         if (skelqh->nextqh)
341             skelqh->nextqh->prevqh = qh;
342         skelqh->nextqh = qh;
343
344         qh->link = skelqh->link;
345         skelqh->link = virt_to_bus(qh) | UHCI_PTR_QH;
346
347         spin_unlock_irqrestore(&uhci->framelist_lock, flags);
348     }
```

参数 skelqh 指向 USB 总线调度系统中的一个节点，这个节点本身也在一个队列中，而挂入这个节点的数据结构又是队列头，即代表着整个控制传输的交互请求队列。所以，节点 skelqh 所在的队列是一个队列的队列，因此称之为“骨架”(skeleton)。USB 的调度系统中有两个骨架，一个是中断交互请求队列的骨架，另一个就是控制(以及成块)传输请求队列的骨架。另一个参数 qh，则指向为具体传输请求建立起来的交互请求队列。同样，uhci_qh 结构也与 USB 控制器硬件的操作有关；所以也是既有采用虚拟地址的链接指针 nextqh 和 prevqh，也有采用物理地址的链接指针 link。至于具体的队列操作，对于读者已经很简单了。这样，代表着本次传输的队列就链入了 USB 总线的调度系统中。

回到 uhci_submit_control( )的代码中，对于全速设备，由于是横向执行，还要调用一个函数 uhci_inc_fsbr( )，使 uhci 结构中的 skel_term_qh，即 skelqh[3]的指针 link，指向 uhci->skel_hs_control_qh。其代码在 drivers/usb/uhci.c 中：

[usb_control_msg( ) > usb_internal_control_msg( ) > usb_start_wait_urb( ) > usb_submit_urb( )
 > uhci_submit_urb( ) > uhci_submit_control( ) > uhci_inc_fsbr( )]

```
563     static void uhci_inc_fsbr(struct uhci *uhci, struct urb *urb)
564     {
565         unsigned long flags;
566         struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;
567
568         if (!urbp)
569             return;
570
571         spin_lock_irqsave(&uhci->framelist_lock, flags);
572
573         if ((!(urb->transfer_flags & USB_NO_FSBR)) && (!urbp->fsbr)) {
574             urbp->fsbr = 1;
575             if (!uhci->fsbr++)
576                 uhci->skel_term_qh.link =
                            virt_to_bus(&uhci->skel_hs_control_qh) | UHCI_PTR_QH;
577         }
578
579         spin_unlock_irqrestore(&uhci->framelist_lock, flags);
580     }
```

队列头 skel_term_qh 是整个调度系统的终点，当 USB 控制器执行到 skel_term_qh 时，当前框架的执行本来已经结束，可以等待下一个框架的开始了。可是，全速设备的控制交互队列是横向执行的，每次只从队列中执行一个交互便转入下一个队列，所以很可能一方面还有交互在等待执行，一方面 USB 控制器却已经空闲。既然如此，何不回过头来再执行全速设备的控制传输队列？在前面 alloc_uhci()的代码中（2229 行），uhci 结构中的成块传输队列头 skel_bulk_qh 的指针 link 已经设置成指向 uhci->skel_term_qh；就是说，在执行完优先级最低的成块传输队列以后，如果本框架中还有空闲时间，就转到“终结”队列头 skel_term_qh。如果这个节点是终点，那么本框架中剩余的时间就放弃了，USB 控制器在这段时间中“空转”等待下一个框架的开始。但是，如果让 skel_term_qh 回过头来链接到 skel_hs_control_qh，则可以在剩余的时间中再回过来执行控制传输队列，直至框架中剩下的时间已不

足以完成一次交互时为止。当然，这只有在存在着采用横向执行的控制传输时才有意义。这里函数名（以及字段名）中的“fsbr”表示“全速设备带宽回收”(Full Speed Bandwidth Reclamation)。如果一个（全速设备的）传输请求没有明确表示不要 FSBR，就将其 fsbr 字段设成 1，并使 uhci 结构中的计数器 fsbr 加 1，表示这个机制多了一个用户。如果原来这个计数器是 0 就将 skel_term_qh 反过来链接到 skel_hs_control_qh。

最后，通过 uhci_add_urb_list( )把 urb 结构链入 uhci 结构的 urb_list。

至此，uhci_submit_control( )的操作已经完成，CPU 经 uhci_submit_urb( )返回到 usb_submit_urb( )，并进而返回到 usb_start_wait_urb( )的代码中，在那里进入睡眠(1019 行)。

如前所述，USB 控制器在每个时间框架中都从等时交互队列开始执行，然后是中断交互。这两种周期性传输(交互)所占的时间合在一起不超过一个框架的 90%，所以在执行完中断交互以后总是还有一些时间可以用来执行控制传输，然后才是成块传输。

具体执行时，USB 控制器因交互请求链入队列的方式而有不同的处理。我们在前面看到，控制（以及成块）交互描述块总是在其所属传输的队列中（指采用物理地址的队列，下同），都有个队列描述块(uhci_qh 结构)。我们称这样的队列为有头队列，并称有头队列内的交互请求是处于一个“队列上下文”中。USB 控制器在其内部维持一个执行堆栈，每碰到一个队列描述块就知道进入了一个新的队列上下文，离开该队列时就回到了此前的上下文。相比之下，等时交互的队列是没有队列头的，中断交互的队列也没有队列头，所以等时交互和中断交互都不在队列上下文中。对于在队列上下文中的交互（请求），USB 控制器在每执行完一个交互以后就自动改变队列头中的指针 element，使其“推进”指向队列中的下一个交互。从而，原来在队列最前面的交互，就是刚执行完毕的交互，就从队列中脱离了出来。所以，USB 控制器在队列上下文中总是执行由指针 element 所指的交互，称为队列的“顶部”。相比之下，如果 USB 控制器不在队列上下文中执行，则在完成一个交互以后并不“推进”指针（实际上也没有指针可以推进），而只是顺着链接指针执行队列中的下一个交互。因此，在这种情况下并不将已经执行的交互从队列中脱链。可想而知，这些交互请求还留在队列中，在下一轮循环中（1 秒钟以后）还会得到执行，这就保证了等时交互和中断交互的周期性。

这里要注意，“完成”当前交互、从而将交互请求中的 TD_CTRL_ACTIVE 标志位改成 0，并在队列上下文中推进指针，跟结束当前交互的执行（从而开始另一个交互的执行或开始空转），是不同的概念。所谓完成一个交互是指：完成了数据的传递并得到确认（如果是输出交互），或者连续发生超时或其他出错，使得继续重发已经失去意义。此时 USB 控制器将当前交互请求的 TD_CTRL_ACTIVE 标志位清成 0，并开始执行队列中的下一个交互请求或转入下一个队列，如果是在队列上下文中则还要推进指针。而结束当前交互的执行，开始下一个交互请求或下一个队列的执行，则并不取决于当前交互的完成。例如，如果对方发回一个 NAK 表示暂时不能完成所要求的传输（如果发生在 SETUP 交互中则相当于一次超时出错），或者因为得不到对方的响应而超时，或者传递的过程中出了错（如 CRC 校验出错），则虽然当前的交互并未完成，但是对这个交互的本次执行也结束了。不同的是，此时不把它的 TD_CTRL_ACTIVE 标志位清 0，也不推进指针，如果是因出错而结束则还要把它的允许重发次数减 1。这样，当 USB 控制器下一次执行到同一队列、同一交互请求时又会再执行一次。

此外，在开始执行一个交互之前，USB 控制器先要判断是否能在本框架内结束这个交互，如果本框架中剩下的时间已经不够则宁可空转，等下一个框架再说。

控制传输的最后一个交互是状态交互。由于这个交互的 TD_CTRL_IOC 标志位为 1，USB 控制器

在这个交互所在的框架结束时就向 CPU 发出一次中断请求。对于 UHCI 接口，USB 总线的中断服务程序为 uhci_interrupt( )，其代码在 drivers/usb/uhci.c 中：

```
2023    static void uhci_interrupt(int irq, void *__uhci, struct pt_regs *regs)
2024    {
2025        struct uhci *uhci = __uhci;
2026        unsigned int io_addr = uhci->io_addr;
2027        unsigned short status;
2028        unsigned long flags;
2029        struct list_head *tmp, *head;
2030
2031        /*
2032         * Read the interrupt status, and write it back to clear the
2033         * interrupt cause
2034         */
2035        status = inw(io_addr + USBSTS);
2036        if (!status)    /* shared interrupt, not mine */
2037            return;
2038        outw(status, io_addr + USBSTS);
2039
2040        if (status & ~(USBSTS_USBINT | USBSTS_ERROR)) {
2041            if (status & USBSTS_RD)
2042                printk(KERN_INFO "uhci: resume detected, not implemented\n");
2043            if (status & USBSTS_HSE)
2044                printk(KERN_ERR "uhci: host system error, PCI problems?\n");
2045            if (status & USBSTS_HCPE)
2046                printk(KERN_ERR
                    "uhci: host controller process error. something bad happened\n");
2047            if (status & USBSTS_HCH) {
2048                printk(KERN_ERR "uhci: host controller halted. very bad\n");
2049                /* FIXME: Reset the controller, fix the offending TD */
2050            }
2051        }
2052
2053        uhci_free_pending_qhs(uhci);
2054
2055        spin_lock(&uhci->urb_remove_lock);
2056        head = &uhci->urb_remove_list;
2057        tmp = head->next;
2058        while (tmp != head) {
2059            struct urb *urb = list_entry(tmp, struct urb, urb_list);
2060
2061            tmp = tmp->next;
2062
2063            list_del(&urb->urb_list);
2064
2065            if (urb->complete)
```

```
2066                urb->complete(urb);
2067        }
2068        spin_unlock(&uhci->urb_remove_lock);
2069
2070        uhci_clear_next_interrupt(uhci);
2071
2072        /* Walk the list of pending TD's to see which ones completed */
2073        nested_lock(&uhci->urblist_lock, flags);
2074        head = &uhci->urb_list;
2075        tmp = head->next;
2076        while (tmp != head) {
2077            struct urb *urb = list_entry(tmp, struct urb, urb_list);
2078
2079            tmp = tmp->next;
2080
2081            /* Checks the status and does all of the magic necessary */
2082            uhci_transfer_result(urb);
2083        }
2084        nested_unlock(&uhci->urblist_lock, flags);
2085    }
```

首先读出 USB 控制器的中断状态寄存器，并将读出的内容写回该寄存器，这是典型的操作。我们先把 2053 行至 2069 行的代码暂搁一下，以后再回过来看。现在先看下面。

前面讲过，USB 控制器在执行中只要碰到一个交互描述块的 TD_CTRL_IOC 标志位为 1，就会在其所在的框架结束时向 CPU 发出中断请求。事实上，即使是一个不"活跃"、因而不需要执行的交互描述块也是一样，USB 控制器会跳过这个交互描述块，但还是会发出中断请求。这样，只要将 skel_term_td 的 TD_CTRL_IOC 标志位设成 1，就实际上每一毫秒就会中断一次。现在既然已经在中断服务程序中，就可以通过 uhci_clear_next_interrupt( )把这个中断源暂时关闭(drivers/usb/uhci.c)。

[uhci_interrupt( ) > uhci_clear_next_interrupt( )]

```
135    void uhci_clear_next_interrupt(struct uhci *uhci)
136    {
137        unsigned long flags;
138
139        spin_lock_irqsave(&uhci->framelist_lock, flags);
140        uhci->skel_term_td.status &= ~TD_CTRL_IOC;
141        spin_unlock_irqrestore(&uhci->framelist_lock, flags);
142    }
143
```

对于正常完成了操作的交互请求，USB 控制器已经自动将其从通过物理地址链接的队列中摘除，因而已经不再在 USB 总线的调度系统中。但是，这些数据结构还在相应 urb 结构（确切地说是 urb_priv 结构）中通过虚拟地址链接的队列中，而 urb 结构又链接在相应 USB 总线的 urb_list 中。现在就要扫描这个 urb 队列，通过 uhci_transfer_result( )将其中已经完成了操作的传输请求找出来，并从队列中摘

除，再“回叫”预先为这些传输安排好的善后操作。这个函数也是一个通用的函数，不仅仅是为控制交互安排的，其代码在 drivers/usb/uhci.c 中：

[uhci_interrupt( ) > uhci_transfer_result( )]

```
1379	/*
1380	 * Return the result of a transfer
1381	 *
1382	 * Must be called with urblist_lock acquired
1383	 */
1384	static void uhci_transfer_result(struct urb *urb)
1385	{
1386		struct usb_device *dev = urb->dev;
1387		struct urb *turb;
1388		int proceed = 0, is_ring = 0;
1389		int ret = -EINVAL;
1390		unsigned long flags;
1391
1392		spin lock_irqsave(&urb->lock, flags);
1393
1394		switch (usb_pipetype(urb->pipe)) {
1395		case PIPE_CONTROL:
1396			ret = uhci_result_control(urb);
1397			break;
1398		case PIPE_INTERRUPT:
1399			ret = uhci_result_interrupt(urb);
1400			break;
1401		case PIPE_BULK:
1402			ret - uhci_result bulk(urb);
1403			break;
1404		case PIPE_ISOCHRONOUS:
1405			ret - uhci_result isochronous(urb);
1406			break;
1407		}
1408
1409		urb->status = ret;
1410
1411		spin_unlock irqrestore(&urb->lock, flags);
1412
1413		if (ret == -EINPROGRESS)
1414			return;
1415
1416		switch (usb_pipetype(urb->pipe)) {
1417		case PIPE_CONTROL:
1418		case PIPE_BULK:
1419		case PIPE_ISOCHRONOUS:
1420			/* Release bandwidth for Interrupt or Isoc. transfers */
```

```
1421            /* Spinlock needed ? */
1422            if (urb->bandwidth)
1423                usb_release_bandwidth(urb >dev, urb, 1);
1424            uhci_unlink_generic(urb);
1425            break;
1426        case PIPE_INTERRUPT:
1427            /* Interrupts are an exception */
1428            if (urb->interval) {
1429                urb->complete(urb);
1430                uhci_reset_interrupt(urb);
1431                return;
1432            }
1433
1434            /* Release bandwidth for Interrupt or Isoc. transfers */
1435            /* Spinlock needed ? */
1436            if (urb->bandwidth)
1437                usb_release_bandwidth(urb->dev, urb, 0);
1438            uhci_unlink_generic(urb);
1439            break;
1440        }
1441
1442        if (urb->next) {
1443            turb = urb->next;
1444            do {
1445                if (turb->status != -EINPROGRESS) {
1446                    proceed = 1;
1447                    break;
1448                }
1449
1450                turb = turb->next;
1451            } while (turb && turb != urb && turb != urb->next);
1452
1453            if (turb == urb || turb == urb->next)
1454                is_ring = 1;
1455        }
1456
1457        if (urb->complete && !proceed) {
1458            urb->complete(urb);
1459            if (!proceed && is_ring)
1460                uhci_submit_urb(urb);
1461        }
1462
1463        if (proceed && urb->next) {
1464            turb = urb->next;
1465            do {
1466                if (turb->status != -EINPROGRESS &&
1467                    uhci_submit_urb(turb) != 0)
1468
```

```
1469                turb = turb->next;
1470            } while (turb && turb != urb->next);
1471
1472            if (urb->complete)
1473                urb->complete(urb);
1474        }
1475
1476        /* We decrement the usage count after we're done with everything */
1477        usb_dec_dev_use(dev);
1478    }
```

参数 urb 指向链接在 urb_list 中的一个 urb 数据结构。显然，对于控制交互首先要对其调用 uhci_result_control( )，这个函数的代码也在 drivers/usb/uhci.c 中：

[uhci_interrupt( ) > uhci_transfer_result( ) > uhci_result_control( )]

```
745    static int uhci_result_control(struct urb *urb)
746    {
747        struct list_head *tmp, *head;
748        struct urb_priv *urbp = urb->hcpriv;
749        struct uhci_td *td;
750        unsigned int status;
751        int ret = 0;
752
753        if (!urbp)
754            return -EINVAL;
755
756        head = &urbp->list;
757        if (head->next == head)
758            return -EINVAL;
759
760        if (urbp->short_control_packet) {
761            tmp = head->prev;
762            goto status_phase;
763        }
764
765        tmp = head->next;
766        td = list_entry(tmp, struct uhci_td, list);
767
768        /* The first TD is the SETUP phase, check the status, but skip */
769        /*  the count */
770        status = uhci_status_bits(td->status);
771        if (status & TD_CTRL_ACTIVE)
772            return -EINPROGRESS;
773
774        if (status)
775            goto td_error;
```

```
776
777         urb->actual_length = 0;
778
779         /* The rest of the TD's (but the last) are data */
780         tmp = tmp->next;
781         while (tmp != head && tmp->next != head) {
782             td = list_entry(tmp, struct uhci_td, list);
783
784             tmp = tmp->next;
785
786             if (urbp->fsbr_timeout && (td->status & TD_CTRL_IOC) &&
787                 !(td->status & TD_CTRL_ACTIVE)) {
788                 uhci_inc_fsbr(urb->dev->bus->hcpriv, urb);
789                 urbp->fsbr_timeout = 0;
790                 td->status &= ~TD_CTRL_IOC;
791             }
792
793             status = uhci_status_bits(td->status);
794             if (status & TD_CTRL_ACTIVE)
795                 return -EINPROGRESS;
796
797             urb->actual_length += uhci_actual_length(td->status);
798
799             if (status)
800                 goto td_error;
801
802             /* Check to see if we received a short packet */
803             if (uhci_actual_length(td->status) < uhci_expected_length(td->info)) {
804                 if (urb->transfer_flags & USB_DISABLE_SPD) {
805                     ret = -EREMOTEIO;
806                     goto err;
807                 }
808
809                 if (uhci_packetid(td->info) == USB_PID_IN)
810                     return usb_control_retrigger_status(urb);
811                 else
812                     return 0;
813             }
814         }
815
816     status_phase:
817         td = list_entry(tmp, struct uhci_td, list);
818
819         /* Control status phase */
820         status = uhci_status_bits(td->status);
821
822     #ifdef I_HAVE_BUGGY_APC_BACKUPS
       ......
```

```
830    #endif
831
832        if (status & TD_CTRL_ACTIVE)
833            return -EINPROGRESS;
834
835        if (status)
836            goto td_error;
837
838        return 0;
839
840    td_error:
841        ret = uhci_map_status(status, uhci_packetout(td->info));
842        if (ret == -EPIPE)
843            /* endpoint has stalled - mark it halted */
844            usb_endpoint_halt(urb->dev, uhci_endpoint(td->info),
845                        uhci_packetout(td->info));
846
847    err:
848        if (debug && ret != -EPIPE) {
849            /* Some debugging code */
850            dbg("uhci_result_control( ) failed with status %x", status);
851
852            /* Print the chain for debugging purposes */
853            uhci_show_urb_queue(urb);
854        }
855
856        return ret;
857    }
```

控制传输有三个阶段，其中第二个阶段有可能为空，所以其交互请求队列中至少应该有两个交互描述块，即使在特殊的情况下（见下）也至少还有最后的状态交互，如果队列为空就错了(757 行)。如果 urbp->short_control_packet 为 1，就说明在传输中曾发生接收到小于应有大小的信包，此时本次传输中只剩下最后一个交互，即状态交互，所以直接跳到状态交互(762 行)，去检查其结果。

在正常的情况下，则扫描本次传输的交互请求队列，顺次检查各个交互描述块。如前所述，USB 总线上控制器会根据执行的结果改变其内容。首先是 SETUP 阶段的交互。如果一个交互的状态位 TD_CTRL_ACTIVE 仍旧是 1，那就说明尚未执行；或者执行失败了，但是失败的次数还在 3 次以下，需要再重新执行。所以返回－EINPROGRESS，表示还在进行中(772 行)。反之，如果 TD_CTRL_ACTIVE 已经变成了 0，那就说明至少 SETUP 阶段的交互已经完成了，此时若状态位中有任何一位为非 0 就表示出了错(775 行)。

下面，就要通过一个 while 循环检查数据阶段的各次交互了。786 行的条件语句目的在于优化。如果原来因为想要回收每个框架尾部的剩余时间而将最后一个队列头链接到了控制传输队列，可是实际上却在相当长一段时间内并没有起到作用，那就说明总线的负载太大了，此时一个由定时器触发的函数 rh_int_timer_do( )就会将为了回收利用剩余时间而作的链接拆除(后面我们还要讲到这个过程)，并把标志位 fsbr_timeout 设成 1。然后，当控制队列中的交互请求得到执行时，总线的负载可能已经轻下来，

所以又可以通过 uhci_inc_fsbr( )恢复 FSBR 链接了。

同样，如果交互描述块中的 TD_CTRL_ACTIVE 标志位为 1 就表示交互尚未完成，或者尚未彻底失败，所以返回-EINPROGRESS。

如果一个交互完成后的数据小于应有的长度(803 行)，那就要看具体的情况。要是在提交该交互请求时把标志位 USB_DISABLE_SPD 设成 1（这里"SPD"是"short packet detect"，即"短信包检测"的意思），那就把该次交互视同失败，所以返回出错代码－EREMOTEIO。否则，如果是输入交互，就通过 usb_control_retrigger_status( )重新调度本次传输，使 USB 总线的主控制器重新执行一遍最后的状态交互。这个函数的代码在 drivers/usb/uhci.c 中：

[uhci_interrupt( ) > uhci_transfer_result( ) > uhci_result_control( ) > usb_control_retrigger_status( )]

```
859    static int usb_control_retrigger_status(struct urb *urb)
860    {
861        struct list_head *tmp, *head;
862        struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;
863        struct uhci *uhci = urb->dev->bus->hcpriv;
864
865        urbp->short_control_packet = 1;
866
867        /* Create a new QH to avoid pointer overwriting problems */
868        uhci_remove_qh(uhci, urbp->qh);
869
870        /* Delete all of the TD's except for the status TD at the end */
871        head = &urbp->list;
872        tmp = head->next;
873        while (tmp != head && tmp->next != head) {
874            struct uhci_td *td = list_entry(tmp, struct uhci_td, list);
875
876            tmp = tmp->next;
877
878            uhci_remove_td_from_urb(urb, td);
879
880            uhci_remove_td(uhci, td);
881
882            uhci_free_td(td);
883        }
884
885        urbp->qh = uhci_alloc_qh(urb->dev);
886        if (!urbp->qh) {
887            err("unable to allocate new QH for control retrigger");
888            return -ENOMEM;
889        }
890
891        /* One TD, who cares about Breadth first? */
892        uhci_insert_tds_in_qh(urbp->qh, urb, 0);
893
```

```
894         /* Low speed or small transfers gets a different queue and treatment */
895         if (urb->pipe & TD_CTRL_LS)
896             uhci_insert_qh(uhci, &uhci->skel_ls_control_qh, urbp->qh);
897         else
898             uhci_insert_qh(uhci, &uhci->skel_hs_control_qh, urbp->qh);
899
900         return -EINPROGRESS;
901     }
```

这里的 865 行把 urbp->short_control_packet 设成 1，并且另外创建一个只包括一个状态交互的队列，使目标设备仍能得到一个确认。这就是前面所说特殊情况的来历。

如果每个数据交互都正常完成，并且最后的状态交互也正常完成，uhci_result_control( )便返回 0。

回到 uhci_transfer_result( )的代码中(drivers/usb/uhci.c，1396 行)，如果从 uhci_result_control( )返回的是－EINPROGRESS，即传输尚未完成，就让本次传输留在所调度的位置上不变而立即返回。否则就要将本次传输从队列中脱离出来，并释放所占用的“带宽”，即占用总线的时间。在 4 种不同的传输中，等时和中断两种是在调度时需要事先为其分配总线带宽的，这样才能保证同一框架中二者之和不超过 90%。如果预先分配了带宽，就一方面记录在(本次传输的)urb 结构中的 bandwidth 字段内，一方面也计入总线的 bus 结构中。对于控制交互，则虽然优先级别比成块传输为高，却并不需要预先分配带宽，因为对成块传输本来就无需保证其带宽。所以，对于控制交互实际上不会调用 usb_release_bandwidth( )。但是，不管是控制、成块还是等时交互，都需要通过 uhci_unlink_generic( )将其 urb 数据结构从各个队列中脱离出来。其代码在 drivers/usb/uhci.c 中：

[uhci_interrupt( ) > uhci_transfer_result( ) > uhci_unlink_generic( )]

```
1480    static int uhci_unlink_generic(struct urb *urb)
1481    {
1482        struct urb_priv *urbp = urb->hcpriv;
1483        struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;
1484
1485        if (!urbp)
1486            return -EINVAL;
1487
1488        uhci_dec_fsbr(uhci, urb);    /* Safe since it checks */
1489
1490        uhci_remove_urb_list(uhci, urb);
1491
1492        if (urbp->qh)
1493            /* The interrupt loop will reclaim the QH's */
1494            uhci_remove_qh(uhci, urbp->qh);
1495
1496        if (!list_empty(&urbp->urb_queue_list))
1497            uhci_delete_queued_urb(uhci, urb);
1498
1499        uhci_destroy_urb_priv(urb);
1500
```

```
1501        urb->dev = NULL;
1502
1503        return 0;
1504    }
```

首先，由于本次传输的完成，对 FSBR，即对于框架末尾剩余时间的回收利用机制，就可能少了一个用户，所以通过 uhci_dec_fsbr( )递减有关的计数，如果此后不再需要此种机制便拆除为而此而建立的链接。然后通过 uhci_remove_urb_list( )，使本次传输的 usb 数据结构脱离 uhci 结构中的队列。这两个函数的代码都很简单，我们就不看了。接着，如果是以传输(而不是交互)为单位调度的，就要通过 uhci_remove_qh( )将 uhci_qh 数据结构从队列中脱链，因为 USB 控制器在执行的过程中会自动将交互描述块脱链，却不会自动将队列描述块脱链。这个函数的代码在 drivers/usb/uhci.c 中：

[uhci_interrupt( ) > uhci_transfer_result( ) > uhci_unlink_generic( ) > uhci_remove_qh( )]

```
350    static void uhci_remove_qh(struct uhci *uhci, struct uhci_qh *qh)
351    {
352        unsigned long flags;
353        int delayed;
354
355        /* If the QH isn't queued, then we don't need to delay unlink it */
356        delayed = (qh->prevqh || qh->nextqh);
357
358        spin_lock_irqsave(&uhci->framelist_lock, flags);
359        if (qh->prevqh) {
360            qh->prevqh->nextqh = qh->nextqh;
361            qh->prevqh->link = qh->link;
362        }
363        if (qh->nextqh)
364            qh->nextqh->prevqh = qh->prevqh;
365        qh->prevqh = qh->nextqh = NULL;
366        qh->element = qh->link = UHCI_PTR_TERM;
367        spin_unlock_irqrestore(&uhci->framelist_lock, flags);
368
369        if (delayed) {
370            spin_lock_irqsave(&uhci->qh_remove_lock, flags);
371
372            /* Check to see if the remove list is empty */
373            /* Set the IOC bit to force an interrupt so we can remove the QH */
374            if (list_empty(&uhci->qh_remove_list))
375                uhci_set_next_interrupt(uhci);
376
377            /* Add it */
378            list_add(&qh->remove_list, &uhci->qh_remove_list);
379
380            spin_unlock_irqrestore(&uhci->qh_remove_lock, flags);
381        } else
```

```
382             uhci_free_qh(qh);
383     }
```

本来，将传输请求从队列中脱链以后就可以释放了。可是，USB 控制器的执行与 CPU 控制器的执行是互相独立的，所以有可能此时 USB 控制器正在这个队列中，所以要先把这个队列头转移到一个单独的队列 qh_remove_list 中，让它“冷却”一下（USB 控制器只能出而不能进），等下一次中断时再来释放。这就是我们在前面跳过的 uhci_free_pending_qhs( )所作的处理(drivers/usb/uhci.c)。

[uhci_interrupt( ) > uhci_free_pending_qhs( )]

```
2002    void uhci_free_pending_qhs(struct uhci *uhci)
2003    {
2004        struct list_head *tmp, *head;
2005        unsigned long flags;
2006
2007        /* Free any pending QH's */
2008        spin_lock_irqsave(&uhci->qh_remove_lock, flags);
2009        head = &uhci->qh_remove_list;
2010        tmp = head->next;
2011        while (tmp != head) {
2012            struct uhci_qh *qh = list_entry(tmp, struct uhci_qh, remove_list);
2013
2014            tmp = tmp->next;
2015
2016            list_del(&qh->remove_list);
2017
2018            uhci_free_qh(qh);
2019        }
2020        spin_unlock_irqrestore(&uhci->qh_remove_lock, flags);
2021    }
```

回到 uhci_unlink_generic( )的代码中，如果 urb_priv 结构中的队列头 urb_queue_list 非空，就说明当初提交传输请求时与其他（对同一设备同一端点的）传输合并了，所以要通过 uhci_delete_queued_urb( ) 将其脱链。其代码在 drivers/usb/uhci.c 中：

[uhci_interrupt( ) > uhci_transfer_result( ) > uhci_unlink_generic( ) > uhci_delete_queued_urb( )]

```
438     static void uhci_delete_queued_urb(struct uhci *uhci, struct urb *urb)
439     {
440         struct urb_priv *urbp, *nurbp;
441         unsigned long flags;
442
443         urbp = urb->hcpriv;
444
445         spin_lock_irqsave(&uhci->append_urb_lock, flags);
446
```

```
447         nurbp = list_entry(urbp->urb_queue_list.next, struct urb_priv,
448                 urb_queue_list);
449 
450         if (!urbp->queued) {
451             /* We're the head, so just insert the QH for the next URB */
452             uhci_insert_qh(uhci, &uhci->skel_bulk_qh, nurbp->qh);
453             nurbp->queued = 0;
454         } else {
455             struct urb_priv *purbp;
456             struct uhci_td *ptd;
457 
458             /* We're somewhere in the middle (or end). A bit trickier */
459             /*  than the head scenario */
460             purbp = list_entry(urbp->urb_queue_list.prev, struct urb_priv,
461                     urb_queue_list);
462 
463             ptd = list_entry(purbp->list.prev, struct uhci_td, list);
464             if (nurbp->queued)
465                 /* Close the gap between the two */
466                 ptd->link = virt_to_bus(list_entry(nurbp->list.next,
467                         struct uhci_td, list));
468             else
469                 /* The next URB happens to be the beggining, so */
470                 /*  we're the last, end the chain */
471                 ptd->link = UHCI_PTR_TERM;
472 
473         }
474 
475         list_del(&urbp->urb_queue_list);
476 
477         spin_unlock_irqrestore(&uhci_append_urb_lock, flags);
478     }
```

最后通过 uhci_destroy_urb_priv( )释放 urb_priv 数据结构，包括队列中的所有 uhci_td 结构，其代码也在 drivers/usb/uhci.c 中：

[uhci_interrupt( ) > uhci_transfer_result( ) > uhci_unlink_generic( ) > uhci_destroy_urb_priv( )]

```
523     static void uhci_destroy_urb_priv(struct urb *urb)
524     {
525         struct list_head *tmp, *head;
526         struct urb_priv *urbp;
527         struct uhci *uhci;
528         struct uhci_td *td;
529         unsigned long flags;
530 
531         spin_lock_irqsave(&urb->lock, flags);
```

```
532
533         urbp = (struct urb_priv *)urb->hcpriv;
534         if (!urbp)
535             goto unlock;
536
537         if (!urb->dev || !urb->dev->bus || !urb->dev->bus->hcpriv)
538             goto unlock;
539
540         uhci = urb->dev->bus->hcpriv;
541
542         head = &urbp->list;
543         tmp = head->next;
544         while (tmp != head) {
545             td = list_entry(tmp, struct uhci_td, list);
546
547             tmp = tmp->next;
548
549             uhci_remove_td_from_urb(urb, td);
550
551             uhci_remove_td(uhci, td);
552
553             uhci_free_td(td);
554         }
555
556         urb->hcpriv = NULL;
557         kmem_cache_free(uhci_up_cachep, urbp);
558
559     unlock:
560         spin_unlock_irqrestore(&urb->lock, flags);
561     }
```

回到 uhci_transfer_result( )的代码中（drivers/usb/uhci.c，1425 行），下面有一段特殊的代码（1442~1474 行）。这段代码主要是为等时传输而设计的，但也可以用于成块传输。

在传输的数据量比较大，又要求在时间上分布得比较均匀时，常常采用“双缓冲”甚至多缓冲的技术。例如对于音频信号，就可以设置两个缓冲区，使这两个缓冲区交替地用于接收数据和处理数据。这样可以使整个流量趋于平均，更为“流水线化”。具体到 USB 总线上的传输，可以为之准备下两个传输请求，即两个 urb 结构，并通过它们的指针 next 互相链接在一起。这样，当一个传输完成，因而要将其从调度系统中脱链时，便通过指针 next 找到其“配偶”，如果不在调度系统中就把它提交调度。这样，如果将这两个传输中的一个老是调度在前 1/2 秒，而另一个老是调度在后 1/2 秒，就可以使流量比只采用一个传输时平均了。至于一般的传输，则由于其指针 next 总是 0，因而代码中的 proceed 与 is_ring 总是 0，所以不受影响。

最后，只要为当前 urb 结构中的函数指针 complete 设置了善后操作，就要调用这个函数。前面在 usb_internal_control_msg( )中把这个函数设置成 usb_api_blocking_completion( )，所以现在就调用这个函数。其代码在 drivers/usb/usb.c 中：

[uhci_interrupt( ) > uhci_transfer_result( ) > usb_api_blocking_completion( )]

```
978     static void usb_api_blocking_completion(urb_t *urb)
979     {
980         api_wrapper_data *awd = (api_wrapper_data *)urb->context;
981
982         if (waitqueue_active(awd->wakeup))
983             wake_up(awd->wakeup);
984     #if 0
985         else
986             dbg("(blocking_completion): waitqueue empty!");
987             // even occurs if urb was unlinked by timeout...
988     #endif
989     }
```

对照一下前面 usb_start_wait_urb( )的代码，就可以看出这里唤醒的正是前面调用了 usb_control_msg( )的那个进程。这个进程当初是通过 schedule_timeout( )睡眠等待(drivers/usb/usb.c，1019 行)的，所以被唤醒的原因可能有两个，即因为操作完成或者因为超时。如果是因超时而被唤醒，而传输实际上已经开始，那就再睡眠。最后，被唤醒的原因还是有两个，要么是传输完成了或者彻底失败了，要么是根本得不到执行而超时。如果是前者，那么 urb 结构已经从调度系统中脱离出来，而如果是后者则 urb 结构仍旧链接在调度系统的队列中，所以先要把它脱离出来(1029 行)。然后，在释放了 urb 数据结构以后，从 usb_start_wait_urb( )返回的 status 指示着传输的成败，而 actual_length 则说明已经传输的实际长度。至于从目标设备读回的数据，则放在预定的缓冲区中。这样，经过 usb_internal_control_msg( )和 usb_control_msg( )逐层返回到 ioctl_scanner( )，本次传输操作已经完成。

我们在前面讲到，就扫描器的控制而言，其应用（驱动）进程可以由系统调用 ioctl( )通过控制传输实现，也可以由一般的 write( )/read( )通过成块传输实现。但是，对于与扫描器间的数据传输，则只能由 write( )/read( )实现。

再看对扫描器的数据读/写，这是通过成块传输完成的。对于扫描器，读操作显然更为重要，所以我们只看系统调用 read( )的实现 read_scanner( )。其代码在 drivers/usb/scanner.c 中：

```
518     static ssize_t
519     read_scanner(struct file * file, char * buffer,
520                  size_t count, loff_t *ppos)
521     {
522         struct scn_usb_data *scn;
523         struct usb_device *dev;
524
525         ssize_t bytes_read; /* Overall count of bytes_read */
526         ssize_t ret;
527
528         kdev_t scn_minor;
529
530         int partial;          /* Number of bytes successfully read */
531         int this_read;        /* Max number of bytes to read */
```

```
532        int result;
533        int rd_expire = RD_EXPIRE;
534
535        char *ibuf;
536
537        scn = file->private_data;
538
539        scn_minor = scn->scn_minor;
540
541        ibuf = scn->ibuf;
542
543        dev = scn->scn_dev;
544
545        bytes_read = 0;
546        ret = 0;
547
548        file->f_dentry->d_inode->i_atime = CURRENT_TIME; /* Update the
549                                                             atime of
550                                                             the device
551                                                             node */
552        down(&(scn->gen_lock));
553
554        while (count > 0) {
555            if (signal_pending(current)) {
556                ret = -EINTR;
557                break;
558            }
559
560            this_read = (count >= IBUF_SIZE) ? IBUF_SIZE : count;
561
562            result = usb_bulk_msg(dev, usb_rcvbulkpipe(dev, scn->bulk_in_ep),
                                     ibuf, this_read, &partial, RD_NAK_TIMEOUT);
563            dbg("read stats(%d): result:%d this_read:%d partial:%d count:%d",
                                  scn_minor, result, this_read, partial, count);
564
565    /*
566     * Scanners are sometimes inheriently slow since they are mechanical
567     * in nature.  USB bulk reads tend to timeout while the scanner is
568     * positioning, resetting, warming up the lamp, etc if the timeout is
569     * set too low.  A very long timeout parameter for bulk reads was used
570     * to overcome this limitation, but this sometimes resulted in folks
571     * having to wait for the timeout to expire after pressing Ctrl-C from
572     * an application. The user was sometimes left with the impression
573     * that something had hung or crashed when in fact the USB read was
574     * just waiting on data.  So, the below code retains the same long
575     * timeout period, but splits it up into smaller parts so that
576     * Ctrl-C's are acted upon in a reasonable amount of time.
577     */
```

```
578
579         if (result == USB_ST_TIMEOUT && !partial) { /* Timeout
580                                                        and no
581                                                        data */
582             if (--rd expire <= 0) {
583                 warn("read_scanner(%d): excessive NAK's received", scn_minor);
584                 ret = -ETIME;
585                 break;
586             } else {
587                 interruptible_sleep_on_timeout(&scn->rd_wait_q, RD_NAK_TIMEOUT);
588                 continue;
589             }
590         } else if ((result < 0) && (result != USB_ST_DATAUNDERRUN)) {
591             warn("read_scanner(%d): funky result:%d. Please notify the maintainer.",
                 scn_minor, (int)result);
592             ret = -EIO;
593             break;
594         }
595
596 #ifdef RD_DATA_DUMP
597         if (partial) {
598             unsigned char cnt, cnt_max;
599             cnt_max = (partial > 24) ? 24 : partial;
600             printk(KERN_DEBUG "dump(%d): ", scn_minor);
601             for (cnt=0; cnt < cnt_max; cnt++) {
602                 printk("%X ", ibuf[cnt]);
603             }
604             printk("\n");
605         }
606 #endif
607
608         if (partial) { /* Data returned */
609             if (copy_to_user(buffer, ibuf, partial)) {
610                 ret = -EFAULT;
611                 break;
612             }
613             count -= this read; /* Compensate for short reads */
614             bytes_read += partial; /* Keep tally of what actually was read */
615             buffer += partial;
616         } else {
617             ret = 0;
618             break;
619         }
620     }
621     up(&(scn->gen_lock));
622
623     return ret ? ret : bytes_read;
624 }
```

我们先把面向扫描器本身的代码留给读者，在这里把注意力集中在对成块传输的调度usb_bulk_msg( )上。由于我们已经详细阅读了有关控制传输的代码，下面只需说明二者的不同之处。

如前所述，每个USB设备的0号端点总是用于控制传输的，并且是双向的端点。除此之外，则设备中的每个“接口”，即逻辑功能都有一组端点，不同类型的传输要使用不同的端点，并且每个端点都是单向的。这些端点的号码都在相应USB设备的枚举阶段通过控制传输取得。取得了一个端点号以后，主机与这个端点之间的就形成了一个逻辑上的“管道”。与控制交互不同，成块传输只有一个阶段，即数据阶段，可以包括一次或多次数据交互。每次交互也是由三个信包构成，第一个总是由主控制器发出的token信包，其中包含着对方的地址和端点号，以及交互的类型，即PID。对于成块传输的PID，include/linux/usb.h中定义了两个宏操作：

```
749     #define usb_sndbulkpipe(dev, endpoint)    \
                    ((PIPE_BULK << 30) | __create_pipe(dev, endpoint))
750     #define usb_rcvbulkpipe(dev, endpoint)    \
                ((PIPE_BULK << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
```

成块传输是通过usb_bulk_msg( )完成的，这个函数的代码在drivers/usb/usb.c中：

[read_scanner( ) > usb_bulk_msg( )]

```
1111    /**
1112     * usb_bulk_msg - Builds a bulk urb, sends it off and waits for completion
1113     * @usb_dev: pointer to the usb device to send the message to
1114     * @pipe: endpoint "pipe" to send the message to
1115     * @data: pointer to the data to send
1116     * @len: length in bytes of the data to send
1117     * @actual_length: pointer to a location to put the actual length
           transfered in bytes
1118     * @timeout: time to wait for the message to complete before timing
           out (if 0 the wait is forever)
1119     *
1120     * This function sends a simple bulk message to a specified endpoint
1121     * and waits for the message to complete, or timeout.
1122     *
1123     * If successful, it returns 0, othwise a negative error number.
1124     * The number of actual bytes transferred will be plaed in the
1125     * actual_timeout paramater.
1126     *
1127     * Don't use this function from within an interrupt context, like a
1128     * bottom half handler.  If you need a asyncronous message, or need to
1129     * send a message from within interrupt context, use usb_submit urb( )
1130     */
1131    int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
1132                void *data, int len, int *actual_length, int timeout)
1133    {
1134        urb_t *urb;
```

```
1135
1136        if (len < 0)
1137            return -EINVAL;
1138
1139        urb=usb_alloc_urb(0);
1140        if (!urb)
1141            return -ENOMEM;
1142
1143        FILL_BULK_URB(urb, usb_dev, pipe, (unsigned char*)data, len,    /* build urb */
1144                (usb_complete_t)usb_api_blocking_completion, 0);
1145
1146        return usb_start_wait_urb(urb, timeout, actual_length);
1147    }
```

对照前面由 usb_control_msg( )调用的 usb_internal_control_msg( )，就可以看出二者几乎一样，只是对成块传输请求的 urb 数据结构是通过 FILL_BULK_URB（而不是 FILL_CONTROL_URB）完成的。这个宏操作定义于 include/linux/usb.h：

```
480    #define FILL_BULK_URB(a, aa, b, c, d, e, f) \
481        do {\
482        spin_lock_init(&(a)->lock);\
483        (a)->dev=aa;\
484        (a)->pipe=b;\
485        (a)->transfer buffer=c;\
486        (a)->transfer_buffer_length=d;\
487        (a)->complete=e;\
488        (a)->context=f;\
489        } while (0)
```

读者不妨比较一下，看看二者有何不同，以及为什么会有不同。

从这以后，一直要到 uhci_submit_urb( )中，成块传输与控制传输才又有不同。我们在这里只列出这个函数中用于成块传输的片段(drivers/usb/uhci.c)：

[read_scanner( ) > usb_bulk_msg( ) > usb_start_wait_urb( ) > usb_submit_urb( ) > uhci_submit_urb( )]

```
        . . . . . .
1342        case PIPE_BULK:
1343            ret = uhci_submit_bulk(urb, u);
1344            break;
        . . . . . .
```

显然，控制传输请求的提交由 uhci_submit_bulk( )完成，其代码在 drivers/usb/uhci.c 中：

[read_scanner( ) > usb_bulk_msg( ) > usb_start_wait_urb( ) > usb_submit_urb( ) > uhci_submit_urb( ) > uhci_submit_bulk( )]

```
1039    static int uhci_submit_bulk(struct urb *urb, struct urb *eurb)
1040    {
1041        struct uhci_td *td;
1042        struct uhci_qh *qh;
1043        unsigned long destination, status;
1044        struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;
1045        int maxsze = usb_maxpacket(urb->dev, urb->pipe, usb_pipeout(urb->pipe));
1046        int len = urb->transfer_buffer_length;
1047        unsigned char *data = urb->transfer_buffer;
1048        struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;
1049
1050        if (len < 0)
1051            return -EINVAL;
1052
1053        /* Can't have low speed bulk transfers */
1054        if (urb->pipe & TD_CTRL_LS)
1055            return -EINVAL;
1056
1057        /* The "pipe" thing contains the destination in bits 8--18 */
1058        destination = (urb->pipe & PIPE_DEVEP_MASK) | usb_packetid(urb->pipe);
1059
1060        /* 3 errors */
1061        status = TD_CTRL_ACTIVE | (3 << TD_CTRL_C_ERR_SHIFT);
1062
1063        if (!(urb->transfer_flags & USB_DISABLE_SPD))
1064            status |= TD_CTRL_SPD;
1065
1066        /*
1067         * Build the DATA TD's
1068         */
1069        do {    /* Allow zero length packets */
1070            int pktsze = len;
1071
1072            if (pktsze > maxsze)
1073                pktsze = maxsze;
1074
1075            td = uhci_alloc_td(urb->dev);
1076            if (!td)
1077                return -ENOMEM;
1078
1079            uhci_add_td_to_urb(urb, td);
1080            uhci_fill_td(td, status, destination | ((pktsze - 1) << 21) |
1081                (usb_gettoggle(urb->dev, usb_pipeendpoint(urb->pipe),
1082                 usb_pipeout(urb->pipe)) << TD_TOKEN_TOGGLE),
1083                virt_to_bus(data));
1084
1085            data += pktsze;
1086            len -= maxsze;
```

```
1087
1088            if (len <= 0)
1089                td->status |= TD_CTRL_IOC;
1090
1091            usb_dotoggle(urb->dev, usb_pipeendpoint(urb->pipe),
1092                usb_pipeout(urb->pipe));
1093        } while (len > 0);
1094
1095        qh = uhci_alloc_qh(urb->dev);
1096        if (!qh)
1097            return -ENOMEM;
1098
1099        urbp->qh = qh;
1100
1101        /* Always assume depth first */
1102        uhci_insert_tds_in_qh(qh, urb, 1);
1103
1104        if (urb->transfer_flags & USB_QUEUE_BULK && eurb) {
1105            urbp->queued = 1;
1106            uhci_append_queued_urb(uhci, eurb, urb);
1107        } else
1108            uhci_insert_qh(uhci, &uhci->skel_bulk_qh, qh);
1109
1110        uhci_add_urb_list(uhci, urb);
1111
1112        uhci_inc_fsbr(uhci, urb);
1113
1114        return -EINPROGRESS;
1115    }
```

读过 uhci_submit_control( )的代码的读者对此自然不会有困难，我们只指出几点。第一，成块传输只能用于全速设备，（试想，如果是低速设备而又要“成块” 传输，怎么能在一个框架，即 1 毫秒的时间里完成？）所以如果是低速设备(1054 行)就立即返回出错代码－EINVAL。第二，成块传输只有一个阶段、一种交互，即数据交互，所以不像控制传输那样还有 SETUP 交互和状态交互。第三，对成块传输队列的执行总是横向，即宽度优先的，所以调用 uhci_insert_tds_in_qh( )时的最后一个参数 breadth 总是 1（1101 行的注释中说“Always assume depth first”正好是说反了）。还有，对成块传输允许将针对同一对象的传输合并（1106 行），所以 uhci_submit_control( )多一个参数 eurb，非 0 时指向已经存在而尚未完成、针对同一对象的成块传输请求。

同样，请求传输的进程也在 usb_start_wait_urb( )中睡眠等待，而 USB 控制器在最后一个交互所在的框架结束时会向 CPU 发出中断请求，USB 总线中断服务程序的入口也同样是 uhci_interrupt( )。不同的是：此时在 uhci_transfer_result( )中调用的是 uhci_result_bulk( )，而不是 uhci_result_control( )。下面是 uhci_transfer_result( )中的一个片段(drivers/usb/uhci.c)：

[uhci_interrupt( ) > uhci_transfer_result( )]

```
        . . . . . .
1401        case PIPE_BULK:
1402            ret = uhci_result_bulk(urb);
1403            break;
        . . . . . .
```

除此以外，就都与控制传输相同了。至于 uhci_result_bulk( )，则在 drivers/usb/uhci.c 中定义为 uhci_result_interrupt( )：

```
1117    /* We can use the result interrupt since they're identical */
1118    #define uhci_result_bulk uhci_result_interrupt
```

实际上，成块传输与中断传输所传递的都是数据，都是可靠传递(如出错就要重发)，只是数据量和(调度)启动的方式不同，所以传输结束时可以由同一个函数加以处理。这个函数的代码在 drivers/usb/uhci.c 中：

[uhci_interrupt( ) > uhci_transfer_result( ) > uhci_result_interrupt( )]

```
940     static int uhci_result_interrupt(struct urb *urb)
941     {
942         struct list_head *tmp, *head;
943         struct urb_priv *urbp = urb->hcpriv;
944         struct uhci_td *td;
945         unsigned int status;
946         int ret = 0;
947
948         if (!urbp)
949             return -EINVAL;
950
951         urb->actual_length = 0;
952
953         head = &urbp->list;
954         tmp = head->next;
955         while (tmp != head) {
956             td = list_entry(tmp, struct uhci_td, list);
957
958             tmp = tmp->next;
959
960             if (urbp->fsbr_timeout && (td->status & TD_CTRL_IOC) &&
961                 !(td->status & TD_CTRL_ACTIVE)) {
962                 uhci_inc_fsbr(urb->dev->bus->hcpriv, urb);
963                 urbp->fsbr_timeout = 0;
964                 td->status &= ~TD_CTRL_IOC;
965             }
966
967             status = uhci_status_bits(td->status);
```

```
968             if (status & TD_CTRL_ACTIVE)
969                 return -EINPROGRESS;
970
971             urb->actual_length += uhci_actual_length(td->status);
972
973             if (status)
974                 goto td_error;
975
976             if (uhci_actual_length(td->status) < uhci_expected_length(td->info)) {
977                 usb_settoggle(urb->dev, uhci_endpoint(td->info),
978                     uhci_packetout(td->info),
979                     uhci_toggle(td->info) ^ 1);
980
981                 if (urb->transfer_flags & USB_DISABLE_SPD) {
982                     ret = -EREMOTEIO;
983                     goto err;
984                 } else
985                     return 0;
986             }
987         }
988
989         return 0;
990
991     td_error:
992         ret = uhci_map_status(status, uhci_packetout(td->info));
993         if (ret == -EPIPE)
994             /* endpoint has stalled - mark it halted */
995             usb_endpoint_halt(urb->dev, uhci_endpoint(td->info),
996                         uhci_packetout(td->info));
997
998     err:
999         if (debug && ret != -EPIPE) {
1000            /* Some debugging code */
1001            dbg("uhci_result_interrupt/bulk( ) failed with status %x",
1002                status);
1003
1004            /* Print the chain for debugging purposes */
1005            if (urbp->qh)
1006                uhci_show_urb_queue(urb);
1007            else
1008                uhci_show_td(td);
1009        }
1010
1011        return ret;
1012    }
```

同样，认真读过 uhci_result_control( )的读者也不应对这段代码感到困难。

读者自然会问：扫描器是速度比较慢的设备，怎么能知道在什么时候去读就有数据可读呢？确实是这样。扫描器(以及其他 USB 设备)在接收到一个 IN 命令(token 信包)时，如果没有数据可以发送就会发送一个 NAK 作为应答。而 USB 控制器，则在数据交互(以及中断交互)中接收到 NAK 时不将交互请求的 TD_CTRL_ACTIVE 位设置成 0，也不推进(本次传输的)队列头中的 element 指针，就好像不曾启动本次交互一样。当然，最后很可能会使相应的传输因超时而失败。所以，从本质上讲，一次 IN 交互(以及 OUT 交互)实际上也是一次查询。那么，为什么不像一般设备驱动那样，让设备在有数据可发送时就主动向主机发出一个中断请求呢？前面讲过，USB 总线的所谓"中断交互"在本质上也是查询，只不过中断交互是周期性的，其执行是有保证的，而成块交互既非周期性也无保证而已。当然，USB 设备只能被动地等待受查询这么一种安排会在一定程度上降低效率，但这是 USB 总线的设计人员在各方面权衡折中以后作出的选择。如果考虑到 USB 总线上相当大比例的流量将是等时传输(不需要查询)，再考虑到总线上设备的个数有限(127)以及总线的速度，并且 USB 控制器本身就带有微处理器，这样的选择应该说是合理的。注意上面所说的超时是指整个传输（而不是某次交互）的超时，在初始化根集中器时设置了一个定时器，定时地通过一个函数 rh_int_timer_do()扫描 USB 总线的 urb_list 队列，将超时的传输请求从队列中摘除。

回到 read_scanner( )的代码中。既然成块传输很可能会因超时而失败，就要把这一点考虑进去。怎么办呢？很简单，只要传输超时，就醒了再睡，过一会儿再来试试(见代码中的 579、587 和 588 行)。也就是说，在较高的层次上也实现定时查询。这样的查询当然不能无限制地进行下去，所以代码中安排了一个计数器 rd_expire。

在扫描器的驱动程序中并未用到中断传输，但是中断传输在对 USB 集中器的操作(从而 USB 设备的热插入)中扮演着重要的角色，所以也应该看一下。我们在前面 usb_hub_configure( )的代码中看到，要调度一个(周期性的)中断传输时，需要为其分配和设置好一个 urb 数据结构，并通过 usb_submit_urb( )提交请求(drivers/usb/hub.c，214～223 行)。

同样，对于采用 UHCI 界面的主控制器，usb_submit_urb( )会调用 uhci_submit_urb( )完成对传输的调度。也正是在这里，对中断传输的调度有了一些与前不同的操作。下面是 uhci_submit_urb( )中的一个片段(drivers/usb/uhci.c)：

[usb_hub_configure( ) > usb_submit_urb( ) > uhci_submit_urb( )]

```
        ......
1329        case PIPE_INTERRUPT:
1330            if (urb->bandwidth == 0) {  /* not yet checked/allocated */
1331                bustime = usb_check_bandwidth(urb->dev, urb);
1332                if (bustime < 0)
1333                    ret = bustime;
1334                else {
1335                    ret = uhci_submit_interrupt(urb);
1336                    if (ret == -EINPROGRESS)
1337                        usb_claim_bandwidth(urb->dev, urb, bustime, 0);
1338                }
1339            } else      /* bandwidth is already set */
1340                ret = uhci_submit_interrupt(urb);
```

```
1341            break;
        . . . . . .
```

中断传输的执行必须是有保证的，所以要预先通过 usb_check_bandwidth( )为之分配“带宽”，即占用总线的时间。这个函数是中断传输和等时传输公用的，其代码在 drivers/usb/usb.c 中：

[usb_hub_configure( ) > usb_submit_urb( ) > uhci_submit_urb( ) > usb_check_bandwidth( )]

```
256    /*
257     * usb_check_bandwidth( ):
258     *
259     * old_alloc is from host_controller->bandwidth_allocated in microseconds;
260     * bustime is from calc_bus_time( ), but converted to microseconds.
261     *
262     * returns <bustime in us> if successful,
263     * or USB_ST BANDWIDTH_ERROR if bandwidth request fails.
264     *
265     * FIXME:
266     * This initial implementation does not use Endpoint.bInterval
267     * in managing bandwidth allocation.
268     * It probably needs to be expanded to use Endpoint.bInterval.
269     * This can be done as a later enhancement (correction).
270     * This will also probably require some kind of
271     * frame allocation tracking...meaning, for example,
272     * that if multiple drivers request interrupts every 10 USB frames,
273     * they don't all have to be allocated at
274     * frame numbers N, N+10, N+20, etc.  Some of them could be at
275     * N+11, N+21, N+31, etc., and others at
276     * N+12, N+22, N+32, etc.
277     * However, this first cut at USB bandwidth allocation does not
278     * contain any frame allocation tracking.
279     */
280    int usb_check_bandwidth (struct usb device *dev, struct urb *urb)
281    {
282        int      new_alloc;
283        int      old_alloc = dev->bus->bandwidth allocated;
284        unsigned int    pipe = urb->pipe;
285        long        bustime;
286
287        bustime = usb_calc_bus_time (usb_pipeslow(pipe), usb_pipein(pipe),
288                usb_pipeisoc(pipe), usb_maxpacket(dev, pipe, usb_pipeout(pipe)));
289        if (usb_pipeisoc(pipe))
290            bustime = NS_TO_US(bustime) / urb->number_of_packets;
291        else
292            bustime = NS_TO_US(bustime);
293
294        new alloc = old_alloc + (int)bustime;
```

```
295             /* what new total allocated bus time would be */
296
297         if (new_alloc > FRAME_TIME_MAX_USECS_ALLOC)
298             dbg("usb-check-bandwidth %sFAILED: was %u, would be %u, bustime = %ld us",
299                 usb_bandwidth_option ? "" : "would have ",
300                 old_alloc, new_alloc, bustime);
301
302         if (!usb_bandwidth_option)  /* don't enforce it */
303             return (bustime);
304         return (new_alloc <= FRAME_TIME_MAX_USECS_ALLOC) ?
                                                bustime : USB_ST_BANDWIDTH_ERROR;
305     }
```

首先，根据目标设备是否为低速设备、传输的方向和类型以及目标设备允许的信包大小（这就是本次传输实际传递的数据量），通过 usb_calc_bus_time( )计算出需要的带宽。这个函数的代码在 drivers/usb/usb.c 中：

[usb_hub_configure( ) > usb_submit_urb( ) > uhci_submit_urb( ) > usb_check_bandwidth( ) > usb_calc_bus_time( )]

```
219     /*
220      * usb_calc_bus_time:
221      *
222      * returns (approximate) USB bus time in nanoseconds for a USB transaction.
223      */
224     static long usb_calc_bus_time (int low_speed, int input_dir, int isoc, int bytecount)
225     {
226         unsigned long   tmp;
227
228         if (low_speed)      /* no isoc. here */
229         {
230             if (input_dir)
231             {
232                 tmp = (67667L * (31L + 10L * BitTime (bytecount))) / 1000L;
233                 return (64060L + (2 * BW_HUB_LS_SETUP) + BW_HOST_DELAY + tmp);
234             }
235             else
236             {
237                 tmp = (66700L * (31L + 10L * BitTime (bytecount))) / 1000L;
238                 return (64107L + (2 * BW_HUB_LS_SETUP) + BW_HOST_DELAY + tmp);
239             }
240         }
241
242         /* for full-speed: */
243
244         if (!isoc)       /* Input or Output */
245         {
```

```
246                tmp = (8354L * (31L + 10L * BitTime (bytecount))) / 1000L;
247                return (9107L + BW_HOST_DELAY + tmp);
248           } /* end not Isoc */
249
250           /* for isoc: */
251
252           tmp = (8354L * (31L + 10L * BitTime (bytecount))) / 1000L;
253           return (((input_dir) ? 7268L : 6265L) + BW_HOST_DELAY + tmp);
254      }
```

具体的计算涉及 USB 总线的物理性质，我们在这里从略，有兴趣或需要的读者可参阅 USB 总线 1.1 版的规格书。计算的结果是以毫微秒为单位的占用总线时间。

回到 usb_check_bandwidth( )中，又通过 NS_TO_US 将毫微秒换算成微秒。如果是等时传输则还要除以需要传递的数据信包数量，因为每个数据信包构成一次交互，即一个调度单位。属于同一传输的不同交互一般依次分布在不同的框架中，而不在同一框架中完成。至于中断传输，则只有一个交互，一个信包。连同原来已经分配的带宽一起，总的带宽不得超过定义于 include/linux/usb.h 的常数 FRAME_TIME_MAX_USECS_ALLOC：

```
795      #define FRAME_TIME_USECS     1000L
796      #define FRAME_TIME_MAX_USECS_ALLOC  (90L * FRAME_TIME_USECS / 100L)
```

就是说，不得超过 1000 微秒的 90%。

为了对框架的容量有个大概的印像，我们在此不妨作一个粗略的估算。对于全速设备，USB 总线的理论带宽是 12Mb，即 1.5MB，除以 1024，则每个框架的带宽大约是 1.5KB，实际上当然达不到这么高。如果每个等时交互的信包大小为 1023 字节，则实际上在每个框架中只能调度一个等时交互；如果信包大小改成 512 字节，那么也只能调度两个等时交互；余可类推。

回到 uhci_submit_urb( )的代码中，如果可以分配所需的带宽，就可以进一步通过 uhci_submit_interrupt( )提交传输请求了。其代码见 drivers/usb/uhci.c。

[usb_hub_configure( ) > usb_submit_urb( ) > uhci_submit_urb( ) > uhci_submit_interrupt( )]

```
906      static int uhci_submit_interrupt(struct urb *urb)
907      {
908           struct uhci_td *td;
909           unsigned long destination, status;
910           struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;
911
912           if (urb->transfer_buffer_length >
                          usb_maxpacket(urb->dev, urb->pipe, usb_pipeout(urb->pipe)))
913                return -EINVAL;
914
915           /* The "pipe" thing contains the destination in bits 8--18 */
916           destination = (urb->pipe & PIPE_DEVEP_MASK) | usb_packetid(urb->pipe);
917
```

```
918         status = (urb->pipe & TD_CTRL_LS) | TD_CTRL_ACTIVE | TD_CTRL_IOC;
919
920         td = uhci_alloc_td(urb->dev);
921         if (!td)
922             return -ENOMEM;
923
924             destination |= (usb_gettoggle(urb->dev, usb_pipeendpoint(urb->pipe),
                                    usb_pipeout(urb->pipe)) << TD_TOKEN_TOGGLE);
925         destination |= ((urb->transfer_buffer_length - 1) << 21);
926
927         usb_dotoggle(urb->dev, usb_pipeendpoint(urb->pipe), usb_pipeout(urb->pipe));
928
929         uhci_add_td_to_urb(urb, td);
930         uhci_fill_td(td, status, destination,
931             virt_to_bus(urb->transfer_buffer));
932
933         uhci_insert_td(uhci, &uhci->skeltd[__interval_to_skel(urb->interval)], td);
934
935         uhci_add_urb_list(uhci, urb);
936
937         return -EINPROGRESS;
938     }
```

这个函数与前面的 uhci_submit_bulk( )很相似，只是成块传输可以有多个数据信包，而中断传输只有一个信包。不过，这二者还有个重要的区别，就是成块传输以传输为调度单位，插入调度队列的是一个代表着成块传输请求的队列(队列头和若干交互请求)；而中断传输则直接把交互请求插入所有中断交互的队列。那么，具体插入到什么位置上呢？以前讲过，数组 skeltd[ ]是中断交互队列的“骨架”，把交互请求插入到这个骨架的哪一点上就决定了中断传输的周期。所以，代码中根据目标设备的中断传输周期通过__interval_to_skel( )计算出插入点的下标，其代码见 drivers/usb/uhci.h。

[usb_hub_configure( ) > usb_submit_urb( ) > uhci_submit_urb( ) > uhci_submit_interrupt( ) > __interval_to_skel( )]

```
253     /*
254      * Search tree for determining where <interval> fits in the
255      * skelqh[ ] skeleton.
256      *
257      * An interrupt request should be placed into the slowest skelqh[ ]
258      * which meets the interval/period/frequency requirement.
259      * An interrupt request is allowed to be faster than <interval> but not slower.
260      *
261      * For a given <interval>, this function returns the appropriate/matching
262      * skelqh[ ] index value.
263      *
264      * NOTE: For UHCI, we don't really need int256_qh since the maximum interval
265      * is 255 ms.  However, we do need an int1_qh since 1 is a valid interval
```

```
266      * and we should meet that frequency when requested to do so.
267      * This will require some change(s) to the UHCI skeleton.
268      */
269     static inline int __interval_to_skel(int interval)
270     {
271         if (interval < 16) {
272             if (interval < 4) {
273                 if (interval < 2)
274                     return 0;   /* int1 for 0-1 ms */
275                 return 1;       /* int2 for 2-3 ms */
276             }
277             if (interval < 8)
278                 return 2;       /* int4 for 4-7 ms */
279             return 3;           /* int8 for 8-15 ms */
280         }
281         if (interval < 64) {
282             if (interval < 32)
283                 return 4;       /* int16 for 16-31 ms */
284             return 5;           /* int32 for 32-63 ms */
285         }
286         if (interval < 128)
287             return 6;           /* int64 for 64-127 ms */
288         return 7;               /* int128 for 128-255 ms (Max.) */
289     }
```

将中断交互请求插入调度队列以后，还要通过 usb_claim_bandwidth( )记下一笔账，以免把同一带宽又重复分配给其他传输。其代码见 drivers/usb/usb.c。

[usb_hub_configure( ) > usb_submit_urb( ) > uhci_submit_urb( ) > uhci_submit_interrupt( ) > usb_claim_bandwidth( )]

```
307     void usb_claim_bandwidth (struct usb_device *dev, struct urb *urb,
                int bustime, int isoc)

308     {
309         dev->bus->bandwidth_allocated += bustime;
310         if (isoc)
311             dev->bus->bandwidth_isoc_reqs++;
312         else
313             dev->bus->bandwidth_int_reqs++;
314         urb->bandwidth = bustime;
315
316     #ifdef USB_BANDWIDTH_MESSAGES
317         dbg("bandwidth alloc increased by %d to %d for %d requesters",
318             bustime,
319             dev->bus->bandwidth_allocated,
320             dev->bus->bandwidth_int_reqs + dev->bus->bandwidth_isoc_reqs);
```

```
321     #endif
322     }
```

传输完成以后，在中断服务程序中会调用 uhci_transfer_result( )。前面我们已经看到，uhci_result_bulk( )实际上也是 uhci_result_interrupt ( )。

对于USB控制器而言，中断交互队列并没有队列头，所以对中断交互的执行不在“队列上下文”中。这样，USB控制器在执行完一个中断交互时就无从推进队列头中的指针（根本就不存在队列头）；从而不会将中断交互从(物理地址)队列中脱链，下一次轮到同一个框架时还会执行这个交互请求。USB控制器的这种特殊设计，与软件上的安排结合在一起，就保证了中断交互(以及等时交互)的周期性。相比之下，控制交互与成块交互在执行完成以后就从(物理地址)队列中脱链了。

对中断交互，在 uhci_transfer_result( )中执行了 uhci_result_interrupt( )以后还有一些特殊的操作，我们再列出有关的代码片断，以便阅读：

[uhci_interrupt( ) > uhci_transfer_result( )]

```
1416        switch (usb_pipetype(urb->pipe)) {
1417        case PIPE_CONTROL:
1418        case PIPE_BULK:
1419        case PIPE_ISOCHRONOUS:
            ......
1425            break;
1426        case PIPE_INTERRUPT:
1427            /* Interrupts are an exception */
1428            if (urb->interval) {
1429                urb->complete(urb);
1430                uhci_reset_interrupt(urb);
1431                return;
1432            }
1433
1434            /* Release bandwidth for Interrupt or Isoc. transfers */
1435            /* Spinlock needed ? */
1436            if (urb->bandwidth)
1437                usb_release_bandwidth(urb->dev, urb, 0);
1438            uhci_unlink_generic(urb);
1439            break;
1440        }
```

中断传输是周期性的，usb 结构中的 interval 字段决定了它的周期。显然，这个周期不应该是 0，所以可以用 0 来表示特殊的意义。事实上，当 interval 为 0 时就表示相应的传输是一次性的(而不是周期性的)操作。所以，代码中(1436～1439 行)当 interval 为 0 时就通过 usb_release_bandwidth( )释放带宽、并且通过 uhci_unlink_generic( )释放有关的数据结构。与其他三种传输在此时的操作(1422~1424 行)相比较，就可以看出完全是一样的。但是，一次性的中断传输毕竟是特殊情况，实际上 interval 总是非 0，否则中断传输就失去了存在的意义。当 interval 非 0 时，一方面先调用事先设置的 complete 函数，唤醒正在睡眠等待的进程，或者向其发送一个信号；另一方面则调用一个函数 uhci_reset_interrupt( )，其代

码在 drivers/usb/uhci.c 中：

[uhci_interrupt( ) > uhci_transfer_result( ) > uhci_reset_interrupt()]

```
1014    static void uhci_reset_interrupt(struct urb *urb)
1015    {
1016        struct list_head *tmp;
1017        struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;
1018        struct uhci_td *td;
1019
1020        if (!urbp)
1021            return;
1022
1023        tmp = urbp->list.next;
1024        td = list_entry(tmp, struct uhci_td, list);
1025        if (!td)
1026            return;
1027
1028        td->status = (td->status & 0x2F000000) | TD_CTRL_ACTIVE | TD_CTRL_IOC;
1029        td->info &= ~(1 << TD_TOKEN_TOGGLE);
1030        td->info |= (usb_gettoggle(urb->dev, usb_pipeendpoint(urb->pipe),
                                 usb_pipeout(urb->pipe)) << TD_TOKEN_TOGGLE);
1031        usb_dotoggle(urb->dev, usb_pipeendpoint(urb->pipe),
                                usb_pipeout(urb->pipe));
1032
1033        urb->status = -EINPROGRESS;
1034    }
```

调用这个函数干什么呢？从代码中可以看出，从这个中断传输的交互队列中找到其（惟一的）交互，将其 TD_CTRL_ACTIVE 标志位(以及 TD_CTRL_IOC 标志位)重新设成 1。此外，也要翻转 tocken 信包中的序号位。前面讲过，USB 控制器在执行中断交互时不将交互描述块脱链，而只是将其 TD_CTRL_ACTIVE 标志位(实际上还有 TD_CTRL_IOC 标志位)清 0，以后执行时就会跳过这个交互。要使 USB 控制器下一次扫描到这个交互描述块时再次执行中断交互，只要把 TD_CTRL_ACTIVE 标志位再设成 1 就行了。所以，如果 interval 非 0，在执行完这个函数后就可以返回了(1431 行)，中断传输的周期性就是这样通过硬件与软件的配合而实现的。如果 interval 为 0，则该次中断传输是“一次性”的，所以完成后要释放占用的带宽并从调度队列中脱链。

看到这里，不知读者是否感到有些异样？试想，等时传输和中断传输一样也是周期性的，理应和中断传输受到同样的处理，怎么反倒与控制传输和成块传输为伍呢？这不是站错了队吗？后面我们会回答这个问题。

扫描器和 USB 集中器都不使用等时传输，那是专为实时的音频以及视频设备（如电话、可视电话等等）而设计的。不过，在阅读了其他三种传输的代码以后，再来看等时传输的代码其实已是水到渠成的事了。我们只把注意力集中在 uhci_submit_urb( )、uhci_submit_isochronous( )、uhci_transfer_result( )、以及 uhci_result_isochronous( )这几个函数上；从驱动程序的角度来说，与其他传输的区别也就在于这

几个函数。顺便提一下，与等时传输有关的函数名和常数定义中常常出现“iso”，这表示 isochronous、而跟“国际标准化组织”毫无关系。

我们先看 uhci_submit_urb( )中的片断(drivers/usb/uhci.c)：

[usb_submit_urb( ) > uhci_submit_urb( )]

```
1345        case PIPE_ISOCHRONOUS:
1346            if (urb->bandwidth == 0) {  /* not yet checked/allocated */
1347                if (urb->number_of_packets <= 0) {
1348                    ret = -EINVAL;
1349                    break;
1350                }
1351                bustime = usb_check_bandwidth(urb->dev, urb);
1352                if (bustime < 0) {
1353                    ret = bustime;
1354                    break;
1355                }
1356
1357                ret = uhci_submit_isochronous(urb);
1358                if (ret == -EINPROGRESS)
1359                    usb_claim_bandwidth(urb->dev, urb, bustime, 1);
1360            } else      /* bandwidth is already set */
1361                ret = uhci_submit_isochronous(urb);
1362            break;
```

显然，这与与用于中断传输的代码几乎完全是一样的，只不过把 uhci_submit_interrupt( )换成了 uhci_submit_isochronous( )。这个函数的代码在 drivers/usb/uhci.c

[usb_submit_urb( ) > uhci_submit_urb( ) > uhci_submit_isochronous( )]

```
1184    static int uhci_submit_isochronous(struct urb *urb)
1185    {
1186        struct uhci_td *td;
1187        struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;
1188        int i, ret, framenum;
1189        int status, destination;
1190
1191        status = TD_CTRL_ACTIVE | TD_CTRL_IOS;
1192        destination = (urb->pipe & PIPE_DEVEP_MASK) | usb_packetid(urb->pipe);
1193
1194        ret = isochronous_find_start(urb);
1195        if (ret)
1196            return ret;
1197
1198        framenum = urb->start_frame;
1199        for (i = 0; i < urb->number_of_packets; i++, framenum++) {
```

```
1200            if (!urb->iso_frame_desc[i].length)
1201                continue;
1202
1203            td = uhci_alloc_td(urb->dev);
1204            if (!td)
1205                return -ENOMEM;
1206
1207            uhci_add_td_to_urb(urb, td);
1208            uhci_fill_td(td, status, destination |
                                    ((urb->iso_frame_desc[i].length - 1) << 21),
1209                virt_to_bus(urb->transfer_buffer + urb->iso_frame_desc[i].offset));
1210
1211            if (i + 1 >= urb->number_of_packets)
1212                td->status |= TD_CTRL_IOC;
1213
1214            uhci_insert_td_frame_list(uhci, td, framenum);
1215        }
```

等时传输中的交互次数完全取决于数据信包的个数。由于只有最后一个信包的 TD_CTRL_IOC 位才设置成 1（见 1211~1212 行），所以要到整个传输结束以后才会引起中断(如果此前没有别的交互引起中断)。

对于 USB 总线，等时信包的内容是无结构的字节流，其各个信包中的数据往往都在同一个大缓冲区中，而只是位移不同，因而不能在各个信包缓冲区之间插入其他信息。所以，要把每个信包缓冲区的起点与长度另外保存在某个地方，为此在 usb 数据结构中设立了一个结构数组 iso_frame_desc[ ]，用来记录本次传输中各个信包缓冲区的起点与长度。等时传输一般是按“帧”进行的，也叫“frame”，这里的“frame_desc”是“帧描述结构”的意思，而并不是指“框架”。具体设备的驱动程序要在调用 usb_submit_urb( )之前设置好这个数组。

与中断传输的调度不同，等时传输要落实到具体的(一个或多个)框架上，构成该等时传输请求的各个交互请求要挂入具体框架的队列中去。前面的 usb_check_bandwidth( )只是从平均带宽的角度确定了能够满足给定等时传输的要求，但是却并没有落实到具体的框架中。所以，代码中通过 isochronous_find_start( )寻找一个框架作为起点，其代码在 drivers/usb/uhci.c 中：

[usb_submit_urb( ) > uhci_submit_urb( ) > uhci_submit_isochronous( ) > isochronous_find_start( )]

```
1158    static int isochronous_find_start(struct urb *urb)
1159    {
1160        int limits;
1161        unsigned int start = 0, end = 0;
1162
1163        if (urb->number_of_packets > 900)    /* 900? Why? */
1164            return -EFBIG;
1165
1166        limits = isochronous_find_limits(urb, &start, &end);
1167
```

```
1168        if (urb->transfer_flags & USB_ISO_ASAP) {
1169            if (limits) {
1170                int curframe;
1171
1172                curframe = uhci_get_current_frame_number(urb->dev) % UHCI_NUMFRAMES;
1173                urb->start_frame = (curframe + 10) % UHCI_NUMFRAMES;
1174            } else
1175                urb->start_frame = end;
1176        } else {
1177            urb->start_frame %= UHCI_NUMFRAMES;
1178            /* FIXME: Sanity check */
1179        }
1180
1181        return 0;
1182    }
```

等时传输的信包大小取决于目标设备，但不得超过 1024 字节。当一个等时传输包含多个信包，从而包含多个交互时，要尽量把这些信包分布到不同的框架中，而不是先把一个框架填满(90%)以后，再前进到下一个框架。进一步，对同一目标设备的不同等时传输也应尽量分布到不同的框架中，要避免使一个框架被同一目标设备的交互使用多次。所以，代码中先通过 isochronous_find_limits( )看一下对于这个等时传输的起点有否限制。这个函数的代码在 drivers/usb/uhci.c 中：

[usb_submit_urb( ) > uhci_submit_urb( ) > uhci_submit_isochronous( ) > isochronous_find_start( ) > isochronous_find_limits( )]

```
1123    static int isochronous_find_limits(struct urb *urb, unsigned int *start,
                                                            unsigned int *end)
1124    {
1125        struct urb *last_urb = NULL;
1126        struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;
1127        struct list_head *tmp, *head = &uhci->urb_list;
1128        int ret = 0;
1129        unsigned long flags;
1130
1131        nested_lock(&uhci->urblist_lock, flags);
1132        tmp = head->next;
1133        while (tmp != head) {
1134            struct urb *u = list_entry(tmp, struct urb, urb_list);
1135
1136            tmp = tmp->next;
1137
1138            /* look for pending URB's with identical pipe handle */
1139            if ((urb->pipe == u->pipe) && (urb->dev == u->dev) &&
1140                (u->status == -EINPROGRESS) && (u != urb)) {
1141                if (!last_urb)
1142                    *start = u->start_frame;
```

```
1143                last_urb = u;
1144            }
1145        }
1146
1147        if (last_urb) {
1148            *end = (last_urb->start_frame + last urb->number_of_packets) & 1023;
1149            ret = 0;
1150        } else
1151            ret = -1;   /* no previous urb found */
1152
1153        nested unlock(&uhci->urblist_lock, flags);
1154
1155        return ret;
1156    }
```

我们把它留给读者。这个函数返回 0 时表示对给定等时传输的起点有限制，返回−1 则表示可以任意决定。回到 isochronous_find_start( )的代码中，标志位 USB_ISO_ASAP 表示“愈早愈好(as soon as possible)”；所以。如果可以任意决定就从 USB 主控器的寄存器中读入当前的框架号，在此基础上加 10 作为起点。不过，由于等时交互的执行是周期性的，所谓早晚只对第一次交互有意义。

回到 uhci_submit_isochronous( )中，可以看出通过 uhci_insert_td_frame_list( )将各个交互请求插入框架时是指定了框架号的，并且框架号逐次递增，其起点就是上面通过 isochronous_find_start( )确定的。这样，就使同一传输的各个交互分布到了不同的框架中。

函数 uhci_insert_td_frame_list( )的代码在 drivers/usb/uhci.c 中，我们把它留给读者。

[usb_submit_urb( ) > uhci_submit_urb( ) > uhci_submit_isochronous( ) > uhci_insert_td_frame_list( )]

```
200    static void uhci_insert_td_frame_list(struct uhci *uhci,
                                          struct uhci_td *td, unsigned framenum)
201    {
202        unsigned long flags;
203        struct uhci_td *nexttd;
204
205        framenum %= UHCI NUMFRAMES;
206
207        spin_lock_irqsave(&uhci->framelist lock, flags);
208
209        td->frameptr = &uhci->fl->frame[framenum];
210        td->link = uhci->fl->frame[framenum];
211        if (!(td->link & (UHCI PTR_TERM | UHCI_PTR QH))) {
212            nexttd = (struct uhci td *)uhci_ptr_to virt(td->link);
213            td->nexttd = nexttd;
214            nexttd->prevtd = td;
215            nexttd->frameptr = NULL;
216        }
217        uhci->fl->frame[framenum] = virt_to_bus(td);
218
```

```
219         spin_unlock_irqrestore(&uhci->framelist_lock, flags);
220     }
```

显然，等时传输的各个交互都是以交互请求为单位插入各个框架，而不像控制交互或成块交互那样是以整个传输为单位将一个队列挂入调度系统中。

USB 控制器对等时交互的执行与中断交互相似。由于等时交互队列物理上并没有队列头，因而等时交互的执行不在“队列上下文”中。这样，主控制器在执行完一个等时交互时便不会推进队列头中的指针，所以不会将等时交互从队列中脱链。

在中断服务程序中调用的 uhci_result_isochronous( )与其他传输大同小异。这个函数的代码在 drivers/usb/uhci.c 中，我们把它留给读者。

[uhci_interrupt( ) > uhci_transfer_result( ) > uhci_result_isochronous( )]

```
1222    static int uhci_result_isochronous(struct urb *urb)
1223    {
1224        struct list_head *tmp, *head;
1225        struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;
1226        int status;
1227        int i, ret = 0;
1228
1229        if (!urbp)
1230            return -EINVAL;
1231
1232        urb->actual_length = 0;
1233
1234        i = 0;
1235        head = &urbp->list;
1236        tmp = head->next;
1237        while (tmp != head) {
1238            struct uhci_td *td = list_entry(tmp, struct uhci_td, list);
1239            int actlength;
1240
1241            tmp = tmp->next;
1242
1243            if (td->status & TD_CTRL_ACTIVE)
1244                return -EINPROGRESS;
1245
1246            actlength = uhci_actual_length(td->status);
1247            urb->iso_frame_desc[i].actual_length = actlength;
1248            urb->actual_length += actlength;
1249
1250            status = uhci_map_status(uhci_status_bits(td->status),
                                                usb_pipeout(urb->pipe));
1251            urb->iso_frame_desc[i].status = status;
1252            if (status != 0) {
1253                urb->error_count++;
```

```
1254                ret = status;
1255            }
1256
1257            i++;
1258        }
1259
1260        return ret;
1261    }
```

但是，就如前面所指出的，uhci_transfer_result( )中对等时传输的处理有些特殊，或者更确切地说，是应该特殊而没有特殊，把周期性的等时传输混同于非周期性的控制传输和成块传输了。我们再来看 uhci_transfer_result( )中的有关片段(drivers/usb/uhci.c)：

[uhci_interrupt( ) > uhci_transfer_result( )]

```
1416        switch (usb_pipetype(urb->pipe)) {
1417        case PIPE_CONTROL:
1418        case PIPE_BULK:
1419        case PIPE_ISOCHRONOUS:
1420            /* Release bandwidth for Interrupt or Isoc. transfers */
1421            /* Spinlock needed ? */
1422            if (urb->bandwidth)
1423                usb_release_bandwidth(urb->dev, urb, 1);
1424            uhci_unlink_generic(urb);
1425            break;
1426        case PIPE_INTERRUPT:
            . . . . . .
1439            break;
1440        }
1441
1442        if (urb->next) {
1443            turb = urb->next;
1444            do {
1445                if (turb->status != -EINPROGRESS) {
1446                    proceed = 1;
1447                    break;
1448                }
1449
1450                turb = turb->next;
1451            } while (turb && turb != urb && turb != urb->next);
1452
1453            if (turb == urb || turb == urb->next)
1454                is_ring = 1;
1455        }
1456
1457        if (urb->complete && !proceed) {
1458            urb->complete(urb);
```

```
1459            if (!proceed && is_ring)
1460                uhci_submit_urb(urb);
1461        }
1462
1463        if (proceed && urb->next) {
1464            turb = urb->next;
1465            do {
1466                if (turb->status != -EINPROGRESS &&
1467                    uhci_submit_urb(turb) != 0)
1468
1469                turb = turb->next;
1470            } while (turb && turb != urb->next);
1471
1472            if (urb->complete)
1473                urb->complete(urb);
1474        }
```

本来，USB 控制器在执行等时交互以后不将其脱链，就是为了保证其周期性。可是，这里仍旧通过 uhci_unlink_generic( )将整个等时传输请求从全部有关的队列中脱链，其中也包括从采用物理地址的队列中脱链，这是在 uhci_unlink_generic( )中经 uhci_destroy_urb_priv( )调用 uhci_remove_td( )完成的，读者可以回过去看一下。也就是说，虽然 USB 控制器的硬件不把等时交互从执行队列中脱链，可是软件还是在中断服务程序中把它脱链了。为什么呢？这要与下面 1442～1474 行的代码结合起来看。前面讲过，这段代码的目的是使等时传输流水线化。我们以电话为例来说明为什么有这个问题。假定有个 IP 电话机接在 USB 总线上，通过等时传输与主机通信，并由主机中的软件和网络接口建立起与对外的连接。为这个目的，在 USB 总线上至少要建立起两个等时传输，分别用于正反两个方向。我们考虑其中从主机到电话机这个方向的等时传输。如果从网上接收到了来自对方的数据，并通过这惟一的一个等时传输发送给电话机，那么电话机每一秒钟才中断一次，每次得到够用一秒钟的数据量。可是，这一来从主机到电话机这一段路上就有了一秒钟的时差。对于有些应用，这固定的一秒钟时差是可以接受的，但是对于电话却是不可接受的。要解决这个问题，就只好把传输的单位分小，例如分成每 50 毫秒一个传输，也就是一秒钟共 20 个传输（尽管是以同一个端点为目标），这样就可以使因为 USB 总线传输而引入的时差降到 200 毫秒。另一方面，即使是可以接受一秒钟时差的应用，也有个从发生中断以后到下一趟执行之间是否来得及处理的问题，所以一般至少也要实行“双缓冲”，即交替使用两个缓冲区的方法。具体到等时传输，就是把整个流量分成两个传输。所以，需要通过等时传输传递的流量总是分成 N 个传输，N 为 1 只是个特例。对于这 N 个传输，usb 结构中提供了一个指针 next，用来把这些传输的 usb 结构连成一个环。每当有一个传输完成而发生中断时，都把这个传输从调度系统中摘下来，供设备驱动程序的高层或应用软件处理，同时就将上一次（或上几次）摘下的传输再提交回调度系统。如此周而复始，就使整个流量分布得均匀了。我们把 1442～1474 行细节留给读者自己阅读。注意凡是已经提交而尚未执行的传输其状态必定是－EINPROGRESS。那么，如果确实只要一个传输就可以了呢？可以让它的指针 next 指出自己，这样在 1460 行就会把它自己再提交给调度系统。

从总体上看，宏观地看，这样的设计当然是很合理（而且很必要）的，但是，从实现的细节看则还可改进。USB 控制器之所以只将在队列上下文中的交互请求从执行队列中摘下，而让不在队列上下文中的交互请求留在队列中，就是因为考虑到了这些交互请求是周期性的。而现在仍要将这些交互请

求反复地摘下又挂上，岂不是没有“把政策用足”？再说，只要不将交互请求中的 TD_CTRL_ACTIVE 标志位设成 1，即使留在队列中也没有关系，USB 控制器在执行时如发现一个交互请求处于“不活跃”状态就会跳过它继续往前执行。还有，交互请求与缓冲区之间的惟一联系只是一个指针 link，真有必要时也还可以“山不转水转”，让交互请求留在队列中而让缓冲区周转。总之，用于等时传输的这段代码是可以优化的。等时传输的实现还是比较新的，还不像内核的主体部分那样已经经历了无数人的横挑鼻子竖挑眼，还没有来得及走过比较深入的优化、改进的阶段，这也是可以理解的。

不过，尽管有待改进，等时传输的功能还是完整的，并已用于具体的设备。

结合以上的解释，读者应该不难读懂与等时传输有关的这几个函数，从而理解等时传输的机理与过程。可是，不结合具体的设备来看这些代码总令人有点“脱离实际”的感觉，所以我们找了一个摄像机的驱动程序作为实例，使读者可以看到具体的设备怎样建立起 USB 总线上的等时传输，又怎样通过等时传输在应用进程与设备之间传递数据。与这种摄像机有关的代码都在 drivers/usb/ibmcam.c 和 drivers/usb/ibmcam.h 两个文件中，我们在这里引用的只是其中的一部分。

同样，摄像机的驱动程序也是通过系统调用 ioctl( )实现对设备的控制，具体的函数为 ibmcam_ioctl( )。与前面的扫描器驱动不同的是，对摄像机的驱动基本上是在内核中实现的，所以其“设备驱动层”比我们前面看到的要厚一些，例如在 ibmcam_ioctl( )中就实现了不少具体的控制命令。但是，我们在这里并不关心摄像机中的具体资源（例如有些什么寄存器）以及对这些资源的具体操作（例如怎样设置色调等等），而以控制传输为手段来实施这些操作这一个“基本点”则是相同的。所以我们把 ibmcam_ioctl( )及有关的代码留给进一步有兴趣的读者。

这里，我们先看等时传输的建立。这是由 ibmcam_init_isoc( )在打开摄像机设备文件时完成的。

[ibmcam_open( ) > ibmcam_init_isoc( )]

```
2148    static int ibmcam_init_isoc(struct usb ibmcam *ibmcam)
2149    {
2150        struct usb_device *dev = ibmcam->dev;
2151        int i, err;
2152
2153        if (!IBMCAM_IS_OPERATIONAL(ibmcam))
2154            return -EFAULT;
2155
2156        ibmcam->compress = 0;
2157        ibmcam->curframe = -1;
2158        ibmcam->cursbuf = 0;
2159        ibmcam->scratchlen = 0;
2160
2161        /* Alternate interface 1 is is the biggest frame size */
2162        i = usb_set_interface(dev, ibmcam->iface, ibmcam->ifaceAltActive);
2163        if (i < 0) {
2164            printk(KERN_ERR "usb_set_interface error\n");
2165            ibmcam->last_error = i;
2166            return -EBUSY;
2167        }
2168        usb_ibmcam_change_lighting_conditions(ibmcam);
```

```
2169        usb_ibmcam_set_sharpness(ibmcam);
2170        usb_ibmcam_reinit_iso(ibmcam, 0);
2171
2172        /* We double buffer the Iso lists */
2173
2174        for (i=0; i < IBMCAM_NUMSBUF; i++) {
2175            int j, k;
2176            urb_t *urb;
2177
2178            urb = usb_alloc_urb(FRAMES_PER_DESC);
2179            if (urb == NULL) {
2180                printk(KERN_ERR "ibmcam_init_isoc: usb_init_isoc( ) failed.\n");
2181                return -ENOMEM;
2182            }
2183            ibmcam->sbuf[i].urb = urb;
2184            urb->dev = dev;
2185            urb->context = ibmcam;
2186            urb->pipe = usb_rcvisocpipe(dev, ibmcam->video_endp);
2187            urb->transfer_flags = USB_ISO_ASAP;
2188            urb->transfer_buffer = ibmcam->sbuf[i].data;
2189            urb->complete = ibmcam_isoc_irq;
2190            urb->number_of_packets = FRAMES_PER_DESC;
2191            urb->transfer_buffer_length = ibmcam->iso_packet_len * FRAMES_PER_DESC;
2192            for (j=k=0; j < FRAMES_PER_DESC; j++, k += ibmcam->iso_packet_len) {
2193                urb->iso_frame_desc[j].offset = k;
2194                urb->iso_frame_desc[j].length = ibmcam->iso_packet_len;
2195            }
2196        }
2197
2198        /* Link URBs into a ring so that they invoke each other infinitely */
2199        for (i=0; i < IBMCAM_NUMSBUF; i++) {
2200            if ((i+1) < IBMCAM_NUMSBUF)
2201                ibmcam->sbuf[i].urb->next = ibmcam->sbuf[i+1].urb;
2202            else
2203                ibmcam->sbuf[i].urb->next = ibmcam->sbuf[0].urb;
2204        }
2205
2206        /* Submit all URBs */
2207        for (i=0; i < IBMCAM_NUMSBUF; i++) {
2208            err = usb_submit_urb(ibmcam->sbuf[i].urb);
2209            if (err)
2210                printk(KERN_ERR "ibmcam_init_isoc: usb_run_isoc(%d) ret %d\n",
2211                       i, err);
2212        }
2213
2214        ibmcam->streaming = 1;
2215        /* printk(KERN_DEBUG "streaming=1 ibmcam->video_endp=$%02x\n",
                                                        ibmcam->video_endp); */
```

```
2216        return 0;
2217    }
```

我们把注意力集中在从 2172～2212 行之间这些代码上。常数 IBMCAM_NUMSBUF 在 drivers/usb/ibmcam.h 中定义为 2，表示采用的是双缓冲，但是显然也可以通过改变其定义而改成多缓冲。对于单向、实时要求又不那么高的视频信号，一般双缓冲就可以了；对交互式的音频信号（如电话）则有更高的要求。代码中通过一个 for 循环为两个缓冲区分别建立起 urb 数据结构，并加以初始化。这里的常数 FRAMES_PER_DESC 定义为 32。从 2186 行可以看出所使用的管道是通过 usb_rcvisocpipe( ) 取得的，因此是输入方向的等时传输。摄像机没有输出方向的等时传输。此外，从 2189 行可见，摄像机的中断服务程序是 ibmcam_isoc_irq( )。

然后，注意 2198～2204 行，这里把这些（两个或更多）urb 数据结构通过结构中的指针 next 连成一个环。

最后，2207～2212 行又通过一个循环提交这些 urb 结构。这样，对于这个摄像机而言，每秒钟将进行两次等时传输，每次包括 32 帧画面，一共是 64 帧，但是也可以通过改变 FRAMES_PER_DESC 的定义加以改变。每当完成其中的一次传输，即 32 帧画面的传递时，就产生一次中断，在上述的 uhci_transfer_result( )中先将刚完成的传输请求脱链，再顺着指针 next 找到上一次已经脱链的传输请求(1445 行)，并再次提交这个传输请求(1467 行)，最后调用摄像机的中断服务程序(1473 行)。

[uhci_interrupt( ) > uhci_transfer_result( ) > ibmcam_isoc_irq( )]

```
1212    static void ibmcam_isoc_irq(struct urb *urb)
1213    {
1214        int len;
1215        struct usb_ibmcam *ibmcam = urb->context;
1216        struct ibmcam_sbuf *sbuf;
1217        int i;
1218
1219        /* We don't want to do anything if we are about to be removed! */
1220        if (!IBMCAM_IS_OPERATIONAL(ibmcam))
1221            return;
1222
1223    #if 0
        ......
1233    #endif
1234
1235        if (!ibmcam->streaming) {
1236            if (debug >= 1)
1237                printk(KERN_DEBUG "ibmcam: oops, not streaming, but interrupt\n");
1238            return;
1239        }
1240
1241        sbuf = &ibmcam->sbuf[ibmcam->cursbuf];
1242
1243        /* Copy the data received into our scratch buffer */
```

```
1244        len = ibmcam_compress_isochronous(ibmcam, urb);
1245
1246        ibmcam->urb_count++;
1247        ibmcam->urb_length = len;
1248        ibmcam->data_count += len;
1249
1250    #if 0
        . . . . . .
1257    #endif
1258
1259        /* If we collected enough data let's parse! */
1260        if (ibmcam->scratchlen) {
1261            /* If we don't have a frame we're current working on, complain */
1262            if (ibmcam->curframe >= 0)
1263                ibmcam_parse_data(ibmcam);
1264            else {
1265                if (debug >= 1)
1266                    printk(KERN_DEBUG
                            "ibmcam: received data, but no frame available\n");
1267            }
1268        }
1269
1270        for (i = 0; i < FRAMES_PER_DESC; i++) {
1271            sbuf->urb->iso_frame_desc[i].status = 0;
1272            sbuf->urb->iso_frame_desc[i].actual_length = 0;
1273        }
1274
1275        /* Move to the next sbuf */
1276        ibmcam->cursbuf = (ibmcam->cursbuf + 1) % IBMCAM_NUMSBUF;
1277
1278        return;
1279    }
```

这里主要的操作是 1244 行对 ibmcam_compress_isochronous( )的调用，这将把从摄像机读入的原始数据复制到另一个缓冲区中。这些原始数据是经过压缩的，所以还要通过 ibmcam_parse_data( )解压缩。解除了压缩的数据就留在缓冲区中，等待应用进程通过系统调用读取。在等时传输中，对缓冲区的写入和读出通常没有互锁和同步，如果应用进程尚未把前一次传输中来自摄像机的数据读走，而后面的数据又来了，就会把前一次的数据覆盖掉。

对摄像机的数据通道只有输入没有输出，所以其系统调用 write( )为空操作，而 read( )的实现则为 ibmcam_read( )。

```
2743    static long ibmcam_read(struct video_device *dev, char *buf,
                                unsigned long count, int noblock)
2744    {
2745        struct usb_ibmcam *ibmcam = (struct usb_ibmcam *)dev;
2746        int frmx = -1;
```

```
2747        volatile struct ibmcam_frame *frame;
2748
2749        if (debug >= 1)
2750            printk(KERN_DEBUG
                    "ibmcam_read: %ld bytes, noblock=%d\n", count, noblock);
2751
2752        if (!IBMCAM_IS_OPERATIONAL(ibmcam) || (buf == NULL))
2753            return -EFAULT;
2754
2755        /* See if a frame is completed, then use it. */
2756        if (ibmcam->frame[0].grabstate >= FRAME_DONE)    /* _DONE or _ERROR */
2757            frmx = 0;
2758        else if (ibmcam->frame[1].grabstate >= FRAME_DONE)/* _DONE or _ERROR */
2759            frmx = 1;
2760
2761        if (noblock && (frmx == -1))
2762            return -EAGAIN;
2763
2764        /* If no FRAME_DONE, look for a FRAME_GRABBING state. */
2765        /* See if a frame is in process (grabbing), then use it. */
2766        if (frmx == -1) {
2767            if (ibmcam->frame[0].grabstate == FRAME_GRABBING)
2768                frmx = 0;
2769            else if (ibmcam->frame[1].grabstate == FRAME_GRABBING)
2770                frmx = 1;
2771        }
2772
2773        /* If no frame is active, start one. */
2774        if (frmx == -1)
2775            ibmcam_new_frame(ibmcam, frmx = 0);
2776
2777        frame = &ibmcam->frame[frmx];
2778
2779    restart:
2780        if (!IBMCAM_IS_OPERATIONAL(ibmcam))
2781            return -EIO;
2782        while (frame->grabstate == FRAME_GRABBING) {
2783            interruptible_sleep_on((void *)&frame->wq);
2784            if (signal_pending(current))
2785                return -EINTR;
2786        }
2787
2788        if (frame->grabstate == FRAME_ERROR) {
2789            frame->bytes_read = 0;
2790            if (ibmcam_new_frame(ibmcam, frmx))
2791                printk(KERN_ERR "ibmcam_read: ibmcam_new_frame error\n");
2792            goto restart;
2793        }
```

```
2794
2795        if (debug >= 1)
2796            printk(KERN_DEBUG
                    "ibmcam_read: frmx=%d, bytes_read=%ld, scanlength=%ld\n",
2797                frmx, frame->bytes_read, frame->scanlength);
2798
2799        /* copy bytes to user space; we allow for partials reads */
2800        if ((count + frame->bytes_read) > frame->scanlength)
2801            count = frame->scanlength - frame->bytes_read;
2802
2803        if (copy_to_user(buf, frame->data + frame->bytes_read, count))
2804            return -EFAULT;
2805
2806        frame->bytes_read += count;
2807        if (debug >= 1)
2808            printk(KERN_DEBUG
                    "ibmcam_read: {copy} count used=%ld, new bytes_read=%ld\n",
2809                count, frame->bytes_read);
2810
2811        if (frame->bytes_read >= frame->scanlength) { /* All data has been read */
2812            frame->bytes_read = 0;
2813
2814            /* Mark it as available to be used again. */
2815            ibmcam->frame[frmx].grabstate = FRAME_UNUSED;
2816            if (ibmcam_new_frame(ibmcam, frmx ? 0 : 1))
2817                printk(KERN_ERR "ibmcam_read: ibmcam_new_frame returned error\n");
2818        }
2819
2820        return count;
2821    }
```

这段代码就留给读者了。

最后，我们还要看一下对传输超时的处理。对控制传输和成块传输的调度是不预先分配带宽的，所以有可能虽然把一个传输（特别是成块传输）挂入了队列，却很长时间不能得到执行。另一方面，我们在前面也提到过，如果目标设备一时不能完成所要求的传输，例如要求从扫描器读入数据，可是扫描器老是因为无数据可读而返回 NAK，则也会造成超时。在这些情况下应该有个机制，让等待时间过长的传输请求因超时而失败。为此，在根集中器的初始化过程中通过 rh_init_int_timer( )设置了一个定时器，使 CPU 每隔一段时间就来扫描一遍 USB 总线的 urb_list，检查各个传输是否已经超时。这个函数的代码在 drivers/usb/uhci.c 中：

```
1774    /* Root Hub INTs are polled by this timer */
1775    static int rh_init_int_timer(struct urb *urb)
1776    {
1777        struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;
1778
1779        uhci->rh.interval = urb->interval;
```

```
1780        init_timer(&uhci->rh.rh_int_timer);
1781        uhci->rh.rh_int_timer.function = rh_int_timer_do;
1782        uhci->rh.rh_int_timer.data = (unsigned long)urb;
1783        uhci->rh.rh_int_timer.expires =
                    jiffies + (HZ * (urb->interval < 30 ? 30 : urb->interval)) / 1000;
1784        add_timer(&uhci->rh.rh_int_timer);
1785
1786        return 0;
1787    }
```

当定时器到点时，CPU 就会执行 rh_int_timer_do()，其代码就在同一文件中：

```
1730    static void rh_int_timer_do(unsigned long ptr)
1731    {
1732        struct urb *urb = (struct urb *)ptr;
1733        struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;
1734        struct list_head *tmp, *head = &uhci->urb_list;
1735        struct urb_priv *urbp;
1736        int len;
1737        unsigned long flags;
1738
1739        if (uhci->rh.send) {
1740            len = rh_send_irq(urb);
1741            if (len > 0) {
1742                urb->actual_length = len;
1743                if (urb->complete)
1744                    urb->complete(urb);
1745            }
1746        }
1747
1748        nested_lock(&uhci->urblist_lock, flags);
1749        tmp = head->next;
1750        while (tmp != head) {
1751            struct urb *u = list_entry(tmp, urb_t, urb_list);
1752
1753            tmp = tmp->next;
1754
1755            urbp = (struct urb_priv *)u->hcpriv;
1756            if (urbp) {
1757                /* Check if the FSBR timed out */
1758                if (urbp->fsbr &&
                        time_after_eq(jiffies, urbp->inserttime + IDLE_TIMEOUT))
1759                    uhci_fsbr_timeout(uhci, u);
1760
1761                /* Check if the URB timed out */
1762                if (u->timeout && time_after_eq(jiffies, u->timeout)) {
1763                    u->transfer_flags |= USB_ASYNC_UNLINK | USB_TIMEOUT_KILLED;
```

```
1764                uhci_unlink_urb(u);
1765            }
1766        }
1767    }
1768    nested_unlock(&uhci->urblist_lock, flags);
1769
1770    rh_init_int_timer(urb);
1771 }
```

这里通过一个 while 循环扫描 uhci 结构中的 urb_list 队列，检查两种超时。一种是 FSBR 超时。如果一个传输请求要求通过回收每个框架中的剩余时间使其尽快得到执行，而实际上却在 IDLE_TIMEOUT，即 50 毫秒以后仍未得到执行，便说明系统的负荷很大，根本就没有剩余时间可以回收，因此通过 uhci_fsbr_timeout( )拆除为此而建立的链接。另一种便是传输本身的超时。读者曾经看到，在提交传输请求时说明了愿意等待的时间。根据当时的时间和愿意等待的时间，就可以计算出在什么时候超时，这就是这里的 u->timeout。如果当前的时间已经过了这一点，那就调用 uhci_unlink_urb( )，将这个传输请求从调度系统中脱离出来，并唤醒正在等待的进程，就好像对这个传输请求的执行已经失败了一样。最后，在完成了扫描以后，又调用 rh_init_int_timer( )再次设置好定时器，使整个过程周而复始。函数 uhci_unlink_urb( )的代码也在 drivers/usb/uhci.c 中，我们就把它留给读者了。

8.10 系统调用 select( )以及异步输入/输出

我们以前讲过，一个已打开文件可以是常规文件，也可以是设备文件，还可以是为进程间通信而建立的管道或插口。当已打开文件是常规文件时，从文件的读出宏观上是同步的，只要尚未读到文件的末尾，虽然启动读操作的进程也可能受阻而进入睡眠，等待从磁盘上读入，但是我们知道这个进程在一个有限的短时期以后一定会被唤醒，这个时期的长短在很大程度上是可预测的。如果已经读到了文件的末尾，则更是立即就可知道。可是，对一些外部设备或进程间通信机制的读出就不同了。就拿键盘来说吧，如果一个应用进程因为读键盘(如 getchar( ))受阻而进入睡眠，那么我们根本无法预测这个进程什么时候会被唤醒，因为我们无法预测操作人员何时会按键盘。从这个意义上说，对设备的读文件操作有可能完全是异步的。对进程间通信机制的读文件操作也是一样。对于这样的已打开文件，我们更倾向于称之为“I/O 通道”。如果一个进程的工作就是等待来自某一个 I/O 通道的输入并作出反应，那就可以用大家熟悉的无穷 while 循环来实现，那就是：启动读操作，如果没有输入数据就睡眠等待，被唤醒就说明有了输入，读取输入数据并作出反应后再启动读操作，如此循环，直至永远。可是，如果输入有可能来自两个或更多个 I/O 通道呢？我们前面提到过一个用来实现“伪终端”的进程就是一个很好的例子：这个进程一方面监视着键盘(包括鼠标器)，一方面监视着一个(或多个)进程间通信管道，无论从哪一个通道有了输入都要马上作出反应。可是，如果应用进程正在睡眠等待键盘输入，而进程间通信管道中却有了数据，则应用进程无法及时读出管道中的数据并作出反应，因为从管道的角度看该进程并不在睡眠等待来自这个管道的输入，从而不会将其喊醒。

在实际应用中，应用软件常常需要同时监视若干个 I/O 通道，等待来自其中任何一个通道的输入数据并作出反应。能够实现这个目标的方法当然是有的，例如让应用进程先通过系统调用 fcntl( )把这些通道的 O_NONBLOCK 标志位设成 1，然后再通过系统调用 read( )读，此时如果没有数据可读就会

立即返回−1，而不会进入睡眠。换言之，就是让应用进程工作于“查询”方式，不断地轮流查询各个通道。这样，从应用进程本身的角度看虽然可以达到目的，但是从系统的角度看却大大降低了系统的效率，在多数情况下是不可接受的。还有个办法是利用进程间通信机制中的报文队列，用一个报文队列来作为输入数据的汇集点。就好像信箱一样：信箱只有一个，信件却可以来自四面八方，全都汇集到一个信箱中(所以在有的系统中把报文队列称作“信箱”)。但是，Linux 内核中的设备驱动程序并不能直接将输入数据转换成一个报文，并投递到报文队列中，所以还得为键盘输入设立一个服务进程。这个进程就监视键盘(包括鼠标器)一个通道，只要有输入就把它转换成报文，并发送到指定的报文队列中。在相反的方面上，对来自进程间通信管道的数据也可以作相似的处理。这样固然也能达到要求，但是系统中多了两个进程，增加了进程调度的负担，也会降低系统的效率(不过不像前一个方法那么严重)。

因此，比较理想的办法是让进程的单一目标的睡眠等待变成多目标的睡眠等待。如果能这样，则在我们这个例子中只要应用进程说明等待的目标是那几个通道就行了。当这几个通道中的任何一个有输入数据时，相应的设备驱动程序就会把睡眠中的进程唤醒。这样，既达到了上述的目标，又保持了较高的效率。

同样的问题也存在于输出操作，因为有时候需要为输出而同时等待若干个通道，等待这些通道中的某一个或几个满足可以进行输出操作的条件。

Linux/Unix 为此提供了一个系统调用 select( )，内核中的实现为 sys_select( )，其代码在 fs/select.c 中：

```
257    asmlinkage long
258    sys_select(int n, fd_set *inp, fd_set *outp, fd_set *exp, struct timeval *tvp)
259    {
260        fd_set_bits fds;
261        char *bits;
262        long timeout;
263        int ret, size;
264
265        timeout = MAX_SCHEDULE_TIMEOUT;
266        if (tvp) {
267            time_t sec, usec;
268
269            if ((ret = verify_area(VERIFY_READ, tvp, sizeof(*tvp)))
270                || (ret = __get_user(sec, &tvp->tv_sec))
271                || (ret = __get_user(usec, &tvp->tv_usec)))
272                goto out_nofds;
273
274            ret = -EINVAL;
275            if (sec < 0 || usec < 0)
276                goto out_nofds;
277
278            if ((unsigned long) sec < MAX_SELECT_SECONDS) {
279                timeout = ROUND_UP(usec, 1000000/HZ);
280                timeout += sec * (unsigned long) HZ;
```

```
281             }
282         }
283 
284         ret = -EINVAL;
285         if (n < 0)
286             goto out_nofds;
287 
288         if (n > current->files->max_fdset)
289             n = current->files->max_fdset;
290 
291         /*
292          * We need 6 bitmaps (in/out/ex for both incoming and outgoing),
293          * since we used fdset we need to allocate memory in units of
294          * long-words.
295          */
296         ret = -ENOMEM;
297         size = FDS_BYTES(n);
298         bits = select_bits_alloc(size);
299         if (!bits)
300             goto out_nofds;
301         fds.in      = (unsigned long *)  bits;
302         fds.out     = (unsigned long *) (bits +   size);
303         fds.ex      = (unsigned long *) (bits + 2*size);
304         fds.res_in  = (unsigned long *) (bits + 3*size);
305         fds.res_out = (unsigned long *) (bits + 4*size);
306         fds.res_ex  = (unsigned long *) (bits + 5*size);
307 
308         if ((ret = get_fd_set(n, inp, fds.in)) ||
309             (ret = get_fd_set(n, outp, fds.out)) ||
310             (ret = get_fd_set(n, exp, fds.ex)))
311             goto out;
312         zero_fd_set(n, fds.res_in);
313         zero_fd_set(n, fds.res_out);
314         zero_fd_set(n, fds.res_ex);
315 
316         ret = do_select(n, &fds, &timeout);
317 
318         if (tvp && !(current->personality & STICKY_TIMEOUTS)) {
319             time_t sec = 0, usec = 0;
320             if (timeout) {
321                 sec = timeout / HZ;
322                 usec = timeout % HZ;
323                 usec *= (1000000/HZ);
324             }
325             put_user(sec, &tvp->tv_sec);
326             put_user(usec, &tvp->tv_usec);
327         }
328 
```

```
329         if (ret < 0)
330             goto out;
331         if (!ret) {
332             ret = -ERESTARTNOHAND;
333             if (signal_pending(current))
334                 goto out;
335             ret = 0;
336         }
337
338         set_fd_set(n, inp, fds.res_in);
339         set_fd_set(n, outp, fds.res_out);
340         set_fd_set(n, exp, fds.res_ex);
341
342     out:
343         select_bits_free(bits, size);
344     out_nofds:
345         return ret;
346     }
```

参数 tvp 指向一个 timeval 数据结构，表明准备睡眠等待的最长时间，如果指针为 0 就表示无限期的睡眠等待；从系统调用返回时，这个数据结构表明还剩下未用完的睡眠时间。参数 inp、outp、exp 都是指向 fd_set 数据结构的指针，这种数据结构是关于已打开文件的位图，位图中的每一位都代表着当前进程的一个已打开文件。早期的系统中都用一个 32 位长字作为已打开文件位图，因为那时候一个进程最多只能有 32 个已打开文件，后来则改成可以灵活定义的 fd_set 数据结构。

调用 sys_select( )时参数 inp 所指的位图表示当前进程在睡眠中要等待来自哪一些已打开文件的输入；返回时则表明哪些已打开文件中已经有了输入。类似地，outp 表示当前进程在睡眠中要等待对哪一些已打开文件的写操作；返回时则表明对哪一些已打开文件的写操作已可立即进行。至于 exp，则用来监视在哪一些通道中发生了异常。最后，参数 n 表示调用时的参数表中有几个位图。所有这些位图以及 timeval 数据结构都在用户空间，所以先要通过__get_user( )和__copy_from_user( )从用户空间把这些数据结构复制到内核中，返回时则要反过来通过 put_user( )和__copy_to_user( )从内核中复制到用户空间。读者对这些代码应该已经熟悉了。

由于 3 个位图各自都有“要求”和“结果”两个版本，在操作的过程中一共需要 6 个位图。所以，先要分配一小块空间用于这 6 个位图，并通过 get_fd_set( )把 3 个“要求”位图从用户空间复制过来(308～310 行)。函数 get_fd_set( )的代码在 include/linux/poll.h 中：

[sys_select( ) > set_fd_set ( )]

```
52     /*
53      * We do a VERIFY_WRITE here even though we are only reading this time:
54      * we'll write to it eventually..
55      *
56      * Use "unsigned long" accesses to let user-mode fd_set's be long-aligned.
57      */
58     static inline
```

```
59     int get_fd_set(unsigned long nr, void *ufdset, unsigned long *fdset)
60     {
61         nr = FDS_BYTES(nr);
62         if (ufdset) {
63             int error;
64             error = verify_area(VERIFY_WRITE, ufdset, nr);
65             if (!error && __copy_from_user(fdset, ufdset, nr))
66                 error = -EFAULT;
67             return error;
68         }
69         memset(fdset, 0, nr);
70         return 0;
71     }
```

操作的主体是 do_select( )，其代码在 fs/select.c 中，我们分段阅读。

[sys_select( ) > do_select( )]

```
163    int do_select(int n, fd_set_bits *fds, long *timeout)
164    {
165        poll_table table, *wait;
166        int retval, i, off;
167        long __timeout = *timeout;
168
169        read_lock(&current->files->file_lock);
170        retval = max_select_fd(n, fds);
171        read_unlock(&current->files->file_lock);
172
173        if (retval < 0)
174            return retval;
175        n = retval;
176
177        poll_initwait(&table);
178        wait = &table;
179        if (!__timeout)
180            wait = NULL;
181        retval = 0;
```

参数 fds 是个指针，指向一个 fd_set_bits 结构，结构中就是 6 个工作位图，其中前 3 个为“要求”位图。170 行通过 max_select_fd( )根据这 3 个位图计算出本次操作所涉及最大的已打开文件号是什么，所有号码高于这个数值的已打开文件都与本次操作无关。

这里用到的数据结构 poll_table 定义于 include/linux/poll.h：

```
15     typedef struct poll_table_struct {
16         int error;
17         struct poll_table_page * table;
18     } poll_table;
```

其主体 poll_table_page 定义于 fs/select.c：

```
27    struct poll_table_entry {
28        struct file * filp;
29        wait_queue_t wait;
30        wait_queue_head_t * wait_address;
31    };
32
33    struct poll_table_page {
34        struct poll_table_page * next;
35        struct poll_table_entry * entry;
36        struct poll_table_entry entries[0];
37    };
```

当一个进程要进入睡眠，而想要某个设备的驱动程序在设备的状态发生变化时将其唤醒，就得要准备好一个 wait_queue_t 数据结构，并将这个数据结构挂入目标设备的某个等待队列中。读者以前曾多次看到，在等待对象单一时一般都把 wait_queue_t 数据结构建立在堆栈中。可是，在有多个等待对象时就不能那样了。另一方面，在有多个等待对象、从而有多个 wait_queue_t 数据结构时，要有个既有效、又灵活，便于扩充的方法将这些 wait_queue_t 结构管理起来，上面这些数据结构就正是为此而设计的。这里的 poll_table_entry 数据结构既是对 wait_queue_t 的扩充，又是对它的“包装”。此外，poll_table_page 结构中数组 entries[ ]的下标为 0，表示该数组的大小可以动态地确定。实际使用时总是分配一个页面，页面中能容纳几个 poll_table_entry 结构，这个数组就是多大。使用中指针 entry 总是指向 entries[ ]中的第一个空闲的 poll_table_entry 结构，根据需要动态地分配 entries[ ]中的表项。一个页面用完了，就再分配一个，通过指针 next 连成一条单链。

函数 do_select( )中定义了一个局部的 poll_table 数据结构 table，177 行先对其进行初始化(include/linux/poll.h)：

[sys_select( ) > do_select( ) > poll_initwait( )]

```
28    static inline void poll_initwait(poll_table* pt)
29    {
30        pt->error = 0;
31        pt->table = NULL;
32    }
```

现在的 poll_table 还只是个空架子，还没有为其分配任何页面，因为还不知道到底需要多少。完成了这些准备以后，CPU 就进入了一个无穷 for 循环，正常情况下一直要到监视中的某个已打开文件中有了输入或满足了其他等待条件，或者指定的睡眠等待时间已经到期，或者当前进程接收到了信号时才会结束。

[sys_select( ) > do_select( )]

```
182        for (;;) {
183            set_current_state(TASK_INTERRUPTIBLE);
```

```
184             for (i = 0 ; i < n; i++) {
185                 unsigned long bit = BIT(i);
186                 unsigned long mask;
187                 struct file *file;
188 
189                 off = i / __NFDBITS;
190                 if (!(bit & BITS(fds, off)))
191                     continue;
192                 file = fget(i);
193                 mask = POLLNVAL;
194                 if (file) {
195                     mask = DEFAULT_POLLMASK;
196                     if (file->f_op && file->f_op->poll)
197                         mask = file->f_op->poll(file, wait);
198                     fput(file);
199                 }
200                 if ((mask & POLLIN_SET) && ISSET(bit, __IN(fds,off))) {
201                     SET(bit, __RES_IN(fds,off));
202                     retval++;
203                     wait = NULL;
204                 }
205                 if ((mask & POLLOUT_SET) && ISSET(bit, __OUT(fds,off))) {
206                     SET(bit, __RES_OUT(fds,off));
207                     retval++;
208                     wait = NULL;
209                 }
210                 if ((mask & POLLEX_SET) && ISSET(bit, __EX(fds,off))) {
211                     SET(bit, __RES_EX(fds,off));
212                     retval++;
213                     wait = NULL;
214                 }
215             }
216             wait = NULL;
217             if (retval || !__timeout || signal_pending(current))
218                 break;
219             if(table.error) {
220                 retval = table.error;
221                 break;
222             }
223             __timeout = schedule_timeout(__timeout);
224         }
225         current->state = TASK_RUNNING;
226 
227         poll_freewait(&table);
228 
229         /*
230          * Up-to-date the caller timeout.
231          */
```

```
232         *timeout = __timeout;
233         return retval;
234     }
```

在这个无穷 for 循环(182～224 行)中，CPU 通过另一个内层 for 循环(184～215 行)对“监视位图”进行一次扫描，这里的宏操作 BITS( )定义于 fs/select.c：

```
112     #define BITS(fds, n)          (*__IN(fds, n)|*__OUT(fds, n)|*__EX(fds, n))
```

如果三个位图之一中的某一位为 1，就对相应的已打开文件作一次询问(197 行)，并把询问的结果汇集到 fds 所指的 fd_set_bits 数据结构中(200～214 行)。一趟扫描下来以后，就检查一下上述的条件是否已经满足，或出了错。如果没有就通过 schedule_timeout( )进入睡眠，到被唤醒时再在下一轮循环中作另一次扫描。就这样，除第一次以外，以后都是在进程被唤醒时才执行一遍循环，所以从本质上讲是一种 do-while 循环。

那么，在什么情况下会被唤醒呢？首先，受到询问的已打开文件的设备驱动程序会把当前进程通过一个 wait_queue_t 数据结构，从而 poll_table_entry 数据结构，挂入其唤醒队列，使得该设备的中断服务程序在接收到输入时就会唤醒这个进程。其次，如果指定了时间限制，则当时间到点时也会唤醒这个进程，这是因为进程在进入睡眠时都指定了需要继续睡眠的时间。最后，如果进程接收到了信号也会被唤醒。

显然，这里的关键在于对具体已打开文件，即设备的询问。从代码中可以看出，这是通过具体 file_operations 数据结构中提供的函数指针 poll 进行的。我们以前都把注意力集中在 open、read、write 等更为常用的操作上，有意忽略了这个函数指针，现在要回过来关注这个操作了。另一方面，阅读 poll 操作的代码在某种意义上也是对有关内容的一次复习。

先看对进程间通信管道的 poll 操作，管道文件读端的 file_operations 数据结构定义于 fs/pipe.c 中，我们再回顾一下：

```
412     struct file_operations read_pipe_fops = {
        . . . . . .
416         poll:       pipe_poll,
        . . . . . .
420     };
```

可见，管道文件读端的 poll 操作是由 pipe_poll( )完成的，其代码也在 fs/pipe.c 中：

[sys_select( ) > do_select( ) > pipe_poll( )]

```
278     /* No kernel lock held - fine */
279     static unsigned int
280     pipe_poll(struct file *filp, poll_table *wait)
281     {
282         unsigned int mask;
283         struct inode *inode = filp->f_dentry->d_inode;
284
```

```
285         poll_wait(filp, PIPE_WAIT(*inode), wait);
286
287         /* Reading only -- no need for acquiring the semaphore.  */
288         mask = POLLIN | POLLRDNORM;
289         if (PIPE_EMPTY(*inode))
290             mask = POLLOUT | POLLWRNORM;
291         if (!PIPE_WRITERS(*inode) && filp->f_version != PIPE_WCOUNTER(*inode))
292             mask |= POLLHUP;
293         if (!PIPE_READERS(*inode))
294             mask |= POLLERR;
295
296         return mask;
297     }
```

管道文件的 inode 结构中有个等待队列的队头，凡是等待着对这个管道进行操作的进程都通过一个 wait_queue_t 数据结构挂入这个队列。当管道的状态发生变化时，若有数据到来或原来满的缓冲区变空时，就会唤醒挂在这个队列中的进程。

宏操作 PIPE_WAIT( )返回这个队头的地址，定义于 include/linux/pipe_fs_i.h:

```
22      #define PIPE_WAIT(inode)     (&(inode).i_pipe->wait)
```

这里的 poll_wait( )是个 inline 函数，定义于 include/linux/poll.h:

[sys_select( ) > do_select( ) > pipe_poll( ) > poll_wait( )]

```
22      extern inline void poll_wait(struct file * filp,
                                 wait_queue_head_t * wait_address, poll_table *p)
23      {
24          if (p && wait_address)
25              __pollwait(filp, wait_address, p);
26      }
```

操作的主体是__pollwait( )，这个函数的代码在 fs/select.c 中：

[sys_select( ) > do_select( ) > pipe_poll( ) > poll_wait( ) > __pollwait( )]

```
74      void __pollwait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
75      {
76          struct poll_table_page *table = p->table;
77
78          if (!table || POLL_TABLE_FULL(table)) {
79              struct poll_table_page *new_table;
80
81              new_table = (struct poll_table_page *) __get_free_page(GFP_KERNEL);
82              if (!new_table) {
83                  p->error = -ENOMEM;
84                     set_current_state(TASK_RUNNING);
```

```
 85                 return;
 86             }
 87             new_table->entry = new_table->entries;
 88             new_table->next = table;
 89             p->table = new_table;
 90             table = new_table;
 91         }
 92 
 93         /* Add a new entry */
 94         {
 95             struct poll_table_entry * entry = table->entry;
 96             table->entry = entry+1;
 97             get_file(filp);
 98             entry->filp = filp;
 99             entry->wait_address = wait_address;
100             init_waitqueue_entry(&entry->wait, current);
101             add_wait_queue(wait_address,&entry->wait);
102         }
103     }
```

如果还没有用于 poll_table_page 结构的页面，或者最后分配的页面已经用完而不再有空闲的 poll_table_entry 结构，就要为其分配一个新的页面，扩充其容量。

然后将 poll_table_page 结构中的下一个 poll_table_entry 数据结构用作连接件，通过它内部的 wait_queue_t 结构 wait 挂入等待队列 wait_address，而这个等待队列的队头正是在目标文件(通道)的 inode 结构中。在 wait_queue_t 结构中包含着指向当前进程 task_struct 结构的指针，这是在 100 行通过 init_waitqueue_entry( )设置的，所以当要将队列中的进程唤醒时从 wait_queue_t 结构开始一下子就能找到相应进程的 task_struct 结构。

以前我们常常看到：当一个进程受阻的时候，一旦把 wait_queue_t 结构挂入一个等待队列，马上就会通过 schedule( )进入睡眠。内核代码中还有一组宏操作 wait_event( )和 wait_event_interruptible( )，把这二者组合在了一起(见第 4 章)，以致人们往往不知不觉地把挂入等待队列与进入睡眠划上了等号。其实，一个进程通过一个 wait_queue_t 结构挂入一个等待队列只是向队列的“主人”挂上一个号，让它在发生某种事件时将其唤醒，而到底是否真的入睡则仍有自由。“唤醒”一个本来就醒着的进程并无什么损害。在 do_select( )操作中，就把这二者区分开来了。这里，每当查询一个通道时，如果这个通道没有数据可读，就都要向该通道挂上一个号，通过 add_wait_queue( )把一个 wait_queue_t 结构挂入该通道的等待对列，但是并不马上进入睡眠；而是要到对所有的通道都查询了一遍以后，回到 do_select( )中，如果确实需要睡眠才会真的进入睡眠。

当然，如果最后并没有睡眠，或者睡眠以后被其中的一个(或几个)通道唤醒，总不能让这些 wait_queue_t 结构留在各个等待队列中，所以 do_select( )中的 227 行要通过 poll_freewait( )把所有这些数据结构从各个队列中摘下来(见后)。

下面我们就来看进程的唤醒，先看当管道的写端进程往管道中写入，从而使得读端有了输入时的情景。读者在第 6 章中读过 pipe_write( )的代码，我们在这里再把其中与唤醒进程有关的片段摘录于下(fs/pipe.c)：

```
134     static ssize_t
135     pipe_write(struct file *filp, const char *buf, size_t count, loff_t *ppos)
136     {
        ......

209             do {
210                 /*
211                  * Synchronous wake-up: it knows that this process
212                  * is going to give up this CPU, so it doesnt have
213                  * to do idle reschedules.
214                  */
215                 wake_up_interruptible_sync(PIPE_WAIT(*inode));
216                 PIPE_WAITING_WRITERS(*inode)++;
217                 pipe_wait(inode);
218                 PIPE_WAITING_WRITERS(*inode)--;
219                 if (signal_pending(current))
220                     goto out;
221                 if (!PIPE_READERS(*inode))
222                     goto sigpipe;
223             } while (!PIPE_FREE(*inode));
224             ret = -EFAULT;
225         }
226
227         /* Signal readers asynchronously that there is more data.  */
228         wake_up_interruptible(PIPE_WAIT(*inode));
        ......
246     }
```

至于具体的唤醒过程，读者已经在第 4 章中阅读过有关的代码，我们就不重复了。这里只是指出，在 pipe_write( )中唤醒的正是在队列 PIPE_WAIT(*inode)中的进程，也就是这个管道文件的 inode 数据结构中的等待队列，与前面读端进程所在的队列是一致的。另一方面，唤醒这个队列中的进程时，从队列中的 wait_queue_t 结构就直接可以找到相应进程的 task_struct 数据结构，而与是否有 poll_table_entry 和 poll_table_page 等数据结构的存在并无关系。

再来看另一个实例，我们假定监视中的另一个通道是鼠标器。同样，也是从这个设备的 file_operations 数据结构 psaux_fops 开始，这个数据结构定义于 drivers/char/pc_keyb.c 中：

```
994     struct file operations psaux_fops = {
995         read:       read_aux,
996         write:      write_aux,
997         poll:       aux_poll,
998         open:       open_aux,
999         release:    release_aux,
1000        fasync:     fasync_aux,
1001    };
```

这个设备的查询操作是由 aux_poll( )完成的，其代码在同一文件(pc_keyb.c)中：

[sys_select( ) > do_select( ) > aux_poll( )]

```
985     /* No kernel lock held - fine */
986     static unsigned int aux_poll(struct file *file, poll_table * wait)
987     {
988         poll_wait(file, &queue->proc_list, wait);
989         if (!queue_empty( ))
990             return POLLIN | POLLRDNORM;
991         return 0;
992     }
```

可见，这里同样也调用 poll_wait( )，不同的是这一次等待队列不在 inode 数据结构中，而在更底层的设备驱动程序中，具体地说是 queue->proc_list。这里的指针 queue 是 drivers/char/pc_keyb.c 中的静态变量，指向一个 aux_queue 数据结构，定义于 include/linux/pc_keyb.h：

```
124     struct aux_queue {
125         unsigned long head;
126         unsigned long tail;
127         wait_queue_head_t proc_list;
128         struct fasync_struct *fasync;
129         unsigned char buf[AUX_BUF_SIZE];
130     };
```

结构中有个成分就是 proc_list，这是个等待队列头，即 wait_queue_head_t 数据结构。所以，把等待队列放在什么地方是由具体的驱动程序自己决定的，只要它自己知道在哪里等待就到哪里去唤醒就可以了。前一个例子中之所以放在 inode 结构中是因为管道没有更底层的设备驱动程序了。那么，鼠标器的驱动程序是否真的到 queue->proc_list 中去唤醒进程呢？当然，下面是 drivers/char/pc_keyb.c 中 handle_mouse_event( )的片段：

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_mouse_event( )]

```
396     static inline void handle_mouse_event(unsigned char scancode)
397     {
398     #ifdef CONFIG_PSMOUSE
        . . . . . .
418             if (head != queue->tail) {
419                 queue->head = head;
420                 kill_fasync(&queue->fasync, SIGIO, POLL_IN);
421                 wake_up_interruptible(&queue->proc_list);
422             }
        . . . . . .
424     #endif
425     }
```

从某种意义上说，do_select( )的策略有点像是"广种薄收"，四处登记要人们在发生什么事时将其唤醒，但是实际上只要有一个通道真的发生了什么而将其唤醒也就可以了。不过，唤醒以后还得对所

有的通道再轮询一遍，因为可能有不止一个通道同时唤醒了这个进程。通过轮询，根据所得到的信息设置 3 个“结果”位图并加以计数，就可以知道到底有几个通道唤醒了它以及各自是为什么原因。这就是 do_select( )中的外层 for 循环的作用。最后，在正常情况下通过 218 行的 break 语句跳出这个无穷 for 循环。

不管有几个通道唤醒了睡眠中的进程，既然唤醒了，不需要再继续睡眠了，就要把所有的 wait_queue_t 结构从各个等待队列中摘下来，这是由 poll_freewait( )完成的。其代码在 fs/select.c 中：

[sys_select( ) > do_select( ) > poll_freewait( )]

```
55    void poll_freewait(poll_table* pt)
56    {
57        struct poll_table_page * p = pt->table;
58        while (p) {
59            struct poll_table_entry * entry;
60            struct poll_table_page *old;
61
62            entry = p->entry;
63            do {
64                entry--;
65                remove_wait_queue(entry->wait_address,&entry->wait);
66                fput(entry->filp);
67            } while (entry > p->entries);
68            old = p;
69            p = p->next;
70            free_page((unsigned long) old);
71        }
72    }
```

我们把这段代码留给读者。通过这段代码，读者应该能体会到为什么要通过 poll_table_entry、poll_table_page、poll_table 等数据结构把所有的 wait_queue_t 结构组装在一起的理由。

回到 sys_select( )的代码中，还要通过 set_fd_set( )把 3 个“结果”位图复制到用户空间（338~340 行）。函数 set_fd_set( )的代码在 include/linux/poll.h 中：

[sys_select( ) > set_fd_set ( )]

```
73    static inline
74    void set_fd_set(unsigned long nr, void *ufdset, unsigned long *fdset)
75    {
76        if (ufdset)
77            __copy_to_user(ufdset, fdset, FDS_BYTES(nr));
78    }
```

最后，通过 select_bits_free( )释放当初分配用于 6 个工作位图的空间。

我们以前讲过，许多外部设备的读操作本质上是异步的，因为我们事先无法估计这些设备在什么

时候有数据可读。另一方面，到现在为止，我们看到的读操作从进程调度的角度看都是同步的，因为当操作受阻的时候进程就进入睡眠等待，这样才能不浪费 CPU 的资源。一般的设备驱动程序就是这两种不同性质操作的交汇点，或者转换点。设备驱动程序的底层是异步的中断机制，上层却是同步的睡眠/唤醒机制。系统调用 select( )把这种转换从一个通道推广到了多个通道，但是并没有改变读操作在顶层的同步性。这样，从整个系统的角度看是避免了效率的降低，但是从进程本身的角度看却未必尽善尽美，因为有时候一个进程也许既想监视若干个通道，同时又想进行一些"后台"计算，只是在监视中的某个通道有事件发生(例如有了输入数据)时才转到"前台"作出反应。那么，有没有办法使顶层，即进程的读文件操作也变成异步呢？其实，进程本来就有类似于中断的异步操作机制，那就是"信号"。所以，只要把上层的信号机制与底层的中断机制结合起来，就能在上层实现对设备的异步操作，Linux 内核提供了这样的功能。

需要对设备的异步操作时，一个进程必须作好下列准备：

(1)　先打开目标设备。

(2)　设置好对目标设备的 SIGIO 信号处理程序，并设置好信号响应(见第 6 章)。

(3)　通过系统调用 fcntl( )将本进程设置成目标通道(已打开文件)的"主人"。

(4)　通过系统调用 ioctl( )将目标通道(已打开文件)设置成异步操作模式。

这样，当目标设备的通道中状态发生变化时，就会向进程发出一个 SIGIO 信号。平时这个进程可以从事"后台"计算，当接收到 SIGIO 信号时就转入前台的信号处理程序中，完成对设备的具体操作。显然，此时的进程就好像受到外部设备中断的 CPU 一样。

下面我们来看这是怎样实现的。首先，file 结构内部有个成分 f_owner，是一个 fown_struct 数据结构，定义于 include/linux/fs.h：

```
492	struct fown_struct {
493		int pid;		/* pid or -pgrp where SIGIO should be sent */
494		uid_t uid, euid;	/* uid/euid of process setting the owner */
495		int signum;		/* posix.1b rt signal to be delivered on IO */
496	};
```

这个数据结构的内容可以通过系统调用 fcntl( )设置，下面是 sys_fcntl( )的主体 do_fcntl( )中的片段(fs/fcntl.c)：

[sys_fcntl( ) > do_fcntl( )]

```
229	static long do_fcntl(unsigned int fd, unsigned int cmd,
230			unsigned long arg, struct file * filp)
231	{
232		long err = -EINVAL;
233
234		switch (cmd) {
247
		. . . . . . .
273		case F_SETOWN:
274			lock_kernel( );
```

```
275                 filp->f_owner.pid = arg;
276                 filp->f_owner.uid = current->uid;
277                 filp->f_owner.euid = current->euid;
278                 err = 0;
279                 if (S_ISSOCK (filp->f_dentry->d_inode->i_mode))
280                     err = sock_fcntl (filp, F_SETOWN, arg);
281                 unlock_kernel( );
282                 break;
        ......
309         }
310
311         return err;
312     }
```

有些设备驱动程序在通过系统调用 ioctl( )设置异步操作模式时会自动设置 f_owner，所以这一步往往并非必须。除此以外，对发送的信号 signum 也可以通过系统调用 fcntl( )另加指定，如果不加指定就是 SIGIO。

再看 sys_ioctl( )的片段，取自文件 fs/ioctl.c：

```
48  asmlinkage long sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
49  {
50      struct file * filp;
51      unsigned int flag;
52      int on, error = -EBADF;
53
54      filp = fget(fd);
        ......
58      lock_kernel( );
59      switch (cmd) {
        ......
83          case FIOASYNC:
84              if ((error = get_user(on, (int *)arg)) != 0)
85                  break;
86              flag = on ? FASYNC : 0;
87
88              /* Did FASYNC state change ? */
89              if ((flag ^ filp->f_flags) & FASYNC) {
90                  if (filp->f_op && filp->f_op->fasync)
91                      error = filp->f_op->fasync(fd, filp, on);
92                  else error = -ENOTTY;
93              }
94              if (error != 0)
95                  break;
96
97              if (on)
98                  filp->f_flags |= FASYNC;
99              else
```

```
100                 filp->f_flags &= ~FASYNC;
101             break;
        . . . . . . .
109         }
110         unlock_kernel( );
111         fput(filp);
112
113     out:
114         return error;
115     }
```

进行 ioctl( )系统调用时，参数 cmd 表示具体的“命令”代码，而 arg 则为对具体命令的参数。对于异步模式的设置，命令码 cmd 为 FIOASYNC，而参数 arg 可以是 1 或 0。

并非所有的设备都支持异步模式。支持异步模式的一般都是与人机界面有关的字符设备，这些设备的 file_operations 结构中的函数指针 fasync 指向其用来建立异步模式的函数。以前面所列鼠标器的 file_operations 结构 psaux_fops 为例，其指针 fasync 指向 fasync_aux( )，它的代码在 drivers/char/pc_keyb.c 中：

[sys_ioctl( ) > fasync_aux( )]

```
860     static int fasync_aux(int fd, struct file *filp, int on)
861     {
862         int retval;
863
864         retval = fasync_helper(fd, filp, on, &queue->fasync);
865         if (retval < 0)
866             return retval;
867         return 0;
868     }
```

这里的全局量 queue 就指向前面看到过的 aux_queue 数据结构，里面有个等待队列 proc_list 用于进程的睡眠/唤醒机制，这我们已经看到过了。但是，这个数据结构中同时还有个 fasync_struct 数据结构指针 fasync，用来维持一个单链的“异步文件”队列(fasync 队列)。函数 fasync_helper( )的作用就是为当前进程创建一个 fasync_struct 数据结构，并将其挂入目标设备的 fasync 队列。这样，当目标设备的通道中发生某些状态变化时，就可以顺着这个队列给每个有关的进程都发一个 SIGIO 信号。这个函数的代码在 fs/fcntl.c 中：

[sys_ioctl( ) > fasync_aux( ) > fasync_helper( )]

```
438     /*
439      * fasync_helper( ) is used by some character device drivers (mainly mice)
440      * to set up the fasync queue. It returns negative on error, 0 if it did
441      * no changes and positive if it added/deleted the entry.
442      */
443     int fasync_helper(int fd, struct file * filp, int on, struct fasync_struct **fapp)
```

```
444 	{
445 		struct fasync_struct *fa, **fp;
446 		struct fasync_struct *new = NULL;
447 		int result = 0;
448 
449 		if (on) {
450 			new = kmem_cache_alloc(fasync_cache, SLAB_KERNEL);
451 			if (!new)
452 				return -ENOMEM;
453 		}
454 		write_lock_irq(&fasync_lock);
455 		for (fp = fapp; (fa = *fp) != NULL; fp = &fa->fa_next) {
456 			if (fa->fa_file == filp) {
457 				if(on) {
458 					fa->fa_fd = fd;
459 					kmem_cache_free(fasync_cache, new);
460 				} else {
461 					*fp = fa->fa_next;
462 					kmem_cache_free(fasync_cache, fa);
463 					result = 1;
464 				}
465 				goto out;
466 			}
467 		}
468 
469 		if (on) {
470 			new->magic = FASYNC_MAGIC;
471 			new->fa_file = filp;
472 			new->fa_fd = fd;
473 			new->fa_next = *fapp;
474 			*fapp = new;
475 			result = 1;
476 		}
477 	out:
478 		write_unlock_irq(&fasync_lock);
479 		return result;
480 	}
```

数据结构(类型)fasync_struct 的定义在 include/linux/fs.h 中：

```
597 	struct fasync_struct {
598 		int magic;
599 		int fa_fd;
600 		struct  fasync_struct   *fa_next; /* singly linked list */
601 		struct  file            *fa_file;
602 	};
```

这个数据结构中并没有指向 task_struct 结构的指针，但是有指向 file 结构的指针；file 结构代表着进程与文件的连接，找到了 file 结构就可以找到它的“主人”。至于 fasync_helper( )的代码，既然这么简单，就不用多说了。

前面，在 handle_mouse_event( )的代码中，我们看到，这个函数一方面通过 wake_up_interruptible( )唤醒等待中的进程，同时还调用了一个函数 kill_fasync( ):

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_mouse_event( )]

```
396     static inline void handle_mouse_event(unsigned char scancode)
397     {
        . . . . . . .
420                 kill_fasync(&queue->fasync, SIGIO, POLL_IN);
        . . . . . . .
425     }
```

函数 kill_fasync( )扫描鼠标器的 fasync 队列，向每个有关的进程发出一个 SIGIO 信号，并将 POLL_IN 传给各个进程的 SIGIO 信号服务程序作为参数，使其知道接收到信号的原因是通道中有了输入。函数 kill_fasync( )的代码在 fs/fcntl.c 中：

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_mouse_event( ) > kill_fasync( )]

```
501     void kill_fasync(struct fasync_struct **fp, int sig, int band)
502     {
503         read_lock(&fasync_lock);
504         __kill_fasync(*fp, sig, band);
505         read_unlock(&fasync_lock);
506     }
```

函数__kill_fasync( )也在同一文件中：

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_mouse_event( ) > kill_fasync( ) > __kill_fasync( )]

```
482     void __kill_fasync(struct fasync_struct *fa, int sig, int band)
483     {
484         while (fa) {
485             struct fown_struct * fown;
486             if (fa->magic != FASYNC_MAGIC) {
487                 printk(KERN_ERR "kill_fasync: bad magic number in "
488                        "fasync_struct!\n");
489                 return;
490             }
491             fown = &fa->fa_file->f_owner;
492             /* Don't send SIGURG to processes which have not set a
493                queued signum: SIGURG has its own default signalling
494                mechanism. */
495             if (fown->pid && !(sig == SIGURG && fown->signum == 0))
```

```
496                     send_sigio(fown, fa->fa_fd, band);
497                 fa = fa->fa_next;
498             }
499     }
```

对于队列中的每一个 fasync_struct 结构，通过与其相联系的 file 结构便可以找到通道的“主人”，这就是需要发送信号的对象。函数 send_sigio( )的代码在 fs/fcntl.c 中：

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_mouse_event( ) > kill_fasync( ) > __kill_fasync( ) > send_sigio( )]

```
413     void send_sigio(struct fown_struct *fown, int fd, int band)
414     {
415         struct task_struct * p;
416         int   pid   = fown->pid;
417
418         read_lock(&tasklist_lock);
419         if ( (pid > 0) && (p = find_task_by_pid(pid)) ) {
420             send_sigio_to_task(p, fown, fd, band);
421             goto out;
422         }
423         for_each_task(p) {
424             int match = p->pid;
425             if (pid < 0)
426                 match =  p->pgrp;
427             if (pid != match)
428                 continue;
429             send_sigio_to_task(p, fown, fd, band);
430         }
431     out:
432         read_unlock(&tasklist_lock);
433     }
```

如果根据对象的 pid 找到了这个进程，就只向这个进程发出信号，否则就向同一进程组中的所有进程都发出信号。函数 send_sigio_to_task( )的代码也在同一文件(fs/fcntl.c)中：

[keyboard_interrupt( ) > handle_kbd_event( ) > handle_mouse_event( ) > kill_fasync( ) > __kill_fasync( ) > send_sigio( ) > send_sigio_to_task( )]

```
374     static void send_sigio_to_task(struct task_struct *p,
375                                 struct fown_struct *fown,
376                                 int fd,
377                                 int reason)
378     {
379         if ((fown->euid != 0) &&
380             (fown->euid ^ p->suid) && (fown->euid ^ p->uid) &&
381             (fown->uid ^ p->suid) && (fown->uid ^ p->uid))
```

```
382             return;
383         switch (fown->signum) {
384             siginfo_t si;
385             default:
386                 /* Queue a rt signal with the appropriate fd as its
387                    value.  We use SI_SIGIO as the source, not
388                    SI_KERNEL, since kernel signals always get
389                    delivered even if we can't queue.  Failure to
390                    queue in this case _should_ be reported; we fall
391                    back to SIGIO in that case. --sct */
392                 si.si_signo = fown->signum;
393                 si.si_errno = 0;
394                     si.si_code  = reason & ~__SI_MASK;
395                 /* Make sure we are called with one of the POLL_*
396                    reasons, otherwise we could leak kernel stack into
397                    userspace.  */
398                 if ((reason & __SI_MASK) != __SI_POLL)
399                     BUG( );
400                 if (reason - POLL_IN >= NSIGPOLL)
401                     si.si_band  = ~0L;
402                 else
403                     si.si_band = band_table[reason - POLL_IN];
404                 si.si_fd    = fd;
405                 if (!send_sig_info(fown->signum, &si, p))
406                     break;
407             /* fall-through: fall back on the old plain SIGIO signal */
408             case 0:
409                 send_sig(SIGIO, p, 1);
410         }
411     }
```

如果没有对发送的信号另加指定，则 fown->signum 为 0，此时发送的信号为 SIGIO，通过 send_sig( ) 发送。如果另加指定则为某种实时信号，通过 send_sig_info( )发送。我们把这段代码留给读者，作为对信号机制的一次复习。

8.11　设备文件系统 devfs

我们以前多次讲到过，以主设备号/次设备号为基础的设备文件管理方式是有根本性的缺点的。这种从 Unix 早期一直延用下来的方案一方面给设备号的管理带来了麻烦，一方面也破坏了/dev 目录的结构。Unix/Linux 系统中所有目录的结构都是层次的，惟独/dev 目录是“平面”的。这不光是风格的问题，也直接影响着访问的效率和管理的方便与否。而且，/dev 目录下的节点并不是按实际的需要而创建的，目录中存在某种设备的节点并不说明内核中有这种设备的驱动程序，更不说明系统中实际连接着这种设备。事实上，几乎所有 Unix/Linux 系统的/dev 目录下都有着大量实际不用的节点。这些节点的存在既降低了效率，又给管理带来麻烦，但是一般又不敢把它们删去，因为一来不知道究竟哪些是

要直接或间接用到的，二来也不知道会不会将来某一天突然需要用到其中的某些节点了。究其原因，问题在于/dev 目录中的节点都是在内核的外部创建的，与内核的构成和运行并无直接的联系。

那么，理想的/dev 目录应该是什么样的呢？首先，它应该是层次的、树状的(不一定是严格意义上的树)。其次，它的规模应该是可伸缩的，而且不受数量的限制(例如 256 个主设备号)。还有，/dev 目录中的内容应该反映系统当前在设备驱动方面的实际情形。例如，这样一套方案就是比较理想的：

(1) 系统加电之初/dev 目录为空。
(2) 系统在初始化阶段扫描并枚举所有连接着的设备，就像对 PCI 总线的扫描枚举一样。每找到一项设备就分门别类地在/dev 目录下创建起子目录，然后以设备的序号作为最底层的节点名，例如“/dev/ide/hd/1”、“/dev/ide/floppy/1”等等。
(3) 以后，每插入一个设备，或安装一个可安装模块，就由内核在/dev 子树中增加一个或几个节点。
(4) 反之，如果关闭或拆除一个设备，或拆除一个可安装模块，就由内核从/dev 子树中删去相应的节点。
(5) 还得与原来的方案兼容。

除将/dev 目录改成树状以外，这里的关键在于将其纳入内核的管理，而不是像以前那样从内核外部管理，那正是造成不一致的原因。如果我们回顾一下前几章中的内容，就可以想起 Linux 其实有一个与此相似的特殊文件系统，那就是/proc。虽然管理的对象和目的不尽相同，但是在方法上显然是可以借鉴的。特殊文件系统 devfs 正是为实现上述目标而设计、与/proc 很相似的“文件系统”。

目前，devfs 的使用还只是一个实验性的选择项，由一个编译选择项 CONFIG_DEVFS_FS 加以选择。

文件系统类型 devfs_fs_type 的定义见 fs/devfs/base.c：

```
3145    static DECLARE_FSTYPE (devfs_fs_type, DEVFS_NAME, devfs_read_super, FS_SINGLE);
```

经过 gcc 的编译预处理以后，就会成为如下的定义：

```
struct file_system_type devfs_fs_type = {
        name:        "devfs",
        read_super:  devfs_read_super,
        fs_flags:    FS_SINGLE,
        owner:       THIS_MODULE,
}
```

系统在初始化时会调用 init_devfs_fs( )进行对 devfs 特殊文件系统的初始化，这个函数的代码在 fs/devfs/base.c 中：

```
3342    int __init init_devfs_fs (void)
3343    {
3344        int err;
3345
```

```
3346        printk ("%s: v%s Richard Gooch (rgooch@atnf.csiro.au)\n",
3347            DEVFS_NAME, DEVFS_VERSION);
3348    #ifdef CONFIG_DEVFS_DEBUG
3349        devfs_debug = devfs_debug_init;
3350        printk ("%s: devfs_debug: 0x%0x\n", DEVFS_NAME, devfs_debug);
3351    #endif
3352        printk ("%s: boot_options: 0x%0x\n", DEVFS_NAME, boot_options);
3353        err = register_filesystem (&devfs_fs_type);
3354        if (!err)
3355        {
3356        struct vfsmount *devfs_mnt = kern_mount (&devfs_fs_type);
3357        err = PTR_ERR (devfs_mnt);
3358        if ( !IS_ERR (devfs_mnt) ) err = 0;
3359        }
3360        return err;
3361    }   /*  End Function init_devfs_fs  */
```

首先通过 register_filesystem( )向系统登记文件系统类型 devfs_fs_type，读者对这个函数应该已经比较熟悉了。然后，就通过 kern_mount( )初始安装特殊文件系统 devfs。读者在第 5 章中看到，通过 kern_mount( )安装的子系统其实并没有纳入以 "/" 为根的文件系统中，暂时还游离在外面。这样的文件系统还要再经过一次常规的安装才能纳入到以 "/" 为根的文件系统中，与其中的某个节点挂上钩，数据结构 devfs_fs_type 的字段 fs_flags 中标志位 FS_SINGLE 为 1 也说明了这一点。事实上，此刻的节点"/"还是空的，对 devfs 的初始安装先于根设备的安装。之所以要在安装根设备之前先初始安装 devfs，是因为在引导系统时可以通过一个命令行选择项指定以哪一个设备作为系统的根设备，所以在安装根设备之前就得要有最低限度的根据设备路径名找到其设备号的功能。

安装的本来意义是将一个设备上的文件系统子树跟已经存在于文件系统中的一个节点挂钩。可是，特殊文件系统实际上并不存在于某个设备上，所以要为其设置一个虚拟的设备。这个设备的主设备号为 UNNAMED_MAJOR，次设备号则由一个函数 get_unnamed_dev( )分配。读者已经在第 5 章中结合/proc 的安装读过 kern_mount( )的代码，知道在初始安装特殊文件系统时要调用一个函数 read_super( )，而 read_super( )，则通过具体 file_system_type 结构中的函数指针 read_super 读入或生成其超级块。从上面数据结构 devfs_fs_type 的定义中可以看到，devfs 的 read_super 操作是 devfs_read_super( )，这个函数的代码在 fs/devfs/base.c 中：

[init_devfs_fs( ) > kern_mount( ) > read_super( ) > devfs_read_super( )]

```
3112    static struct super_block *devfs_read_super (struct super_block *sb,
3113                                void *data, int silent)
3114    {
3115        struct inode *root_inode = NULL;
3116
3117        if (get_root_entry ( ) == NULL) goto out_no_root;
3118        atomic_set (&fs_info.devfsd_overrun_count, 0);
3119        init_waitqueue_head (&fs_info.devfsd_wait_queue);
3120        init_waitqueue_head (&fs_info.revalidate_wait_queue);
```

```
3121        fs_info.sb = sb;
3122        sb->u.generic_sbp = &fs_info;
3123        sb->s_blocksize = 1024;
3124        sb->s_blocksize_bits = 10;
3125        sb->s_magic = DEVFS_SUPER_MAGIC;
3126        sb->s_op = &devfs_sops;
3127        if ( ( root_inode = get_vfs_inode (sb, root_entry, NULL) ) == NULL )
3128        goto out_no_root;
3129        sb->s_root = d_alloc_root (root_inode);
3130        if (!sb->s_root) goto out_no_root;
3131    #ifdef CONFIG_DEVFS_DEBUG
3132        if (devfs_debug & DEBUG_DISABLED)
3133        printk ("%s: read super, made devfs ptr: %p\n",
3134            DEVFS_NAME, sb->u.generic_sbp);
3135    #endif
3136        return sb;
3137
3138    out_no_root:
3139        printk ("devfs_read_super: get root inode failed\n");
3140        if (root_inode) iput (root_inode);
3141        return NULL;
3142    }   /*  End Function devfs_read_super  */
```

像/proc 一样，devfs 在磁盘上也没有对应物，不像常规的文件系统那样在磁盘上有连接成树状的目录节点和索引节点，所以在内存中要为之建立起一套连接成树状的数据结构。对于 devfs 文件系统，这种数据结构是 devfs_entry，定义于 fs/devfs/base.c：

```
630     struct devfs_entry
631     {
632         void *info;
633         union
634         {
635         struct directory_type dir;
636         struct fcb_type fcb;
637         struct symlink_type symlink;
638         struct fifo_type fifo;
639         }
640         u;
641         struct devfs_entry *prev; /* Previous entry in the parent directory  */
642         struct devfs_entry *next; /*  Next entry in the parent directory     */
643         struct devfs_entry *parent;  /*  The parent directory               */
644         struct devfs_entry *slave;   /*  Another entry to unregister        */
645         struct devfs_inode inode;
646         umode_t mode;
647         unsigned short namelen;  /* I think 64k+ filenames are a way off... */
648         unsigned char registered:1;
649         unsigned char show_unreg:1;
```

```
650         unsigned char hide:1;
651         unsigned char no_persistence:1;
652         char name[1];     /*  This is just a dummy: the allocated array is
653                      bigger. This is NULL-terminated  */
654     };
```

结构中的字符数组 name[ ]就是节点名，其大小在为具体的数据结构分配空间时根据具体的字符串长度确定。每个 devfs_entry 结构通过指针 prev、next、parent 和 slave 连接成树状，以此来实现一个文件系统子树。从数据结构的定义可以看出，devfs 子树中的节点有四种不同的类型。第一种是 dir，即目录，这是不言自明的。第二种是 fcb，即“文件控制块”，这是 devfs 子树中的叶节点。第三种是 symlink，这也是不言自明的。最后一种是 fifo，专门用于管道文件。因节点类型的不同，devfs_entry 结构中的 union u 也相应地解释成不同的数据结构。这里先看一下 fcb_type 数据结构的定义(fs/devfs/base.c)：

```
577     struct file_type
578     {
579         unsigned long size;
580     };
581
582     struct device_type
583     {
584         unsigned short major;
585         unsigned short minor;
586     };
587
588     struct fcb_type  /*  File, char, block type  */
589     {
590         uid_t default_uid;
591         gid_t default_gid;
592         void *ops;
593         union
594         {
595         struct file_type file;
596         struct device_type device;
597         }
598         u;
599         unsigned char auto_owner:1;
600         unsigned char aopen_notify:1;
601         unsigned char removable:1;/* Belongs in device_type, but save space */
602         unsigned char open:1;      /*  Not entirely correct                  */
603     };
```

可见，devfs 中的叶节点有两种类型，一种是文件，另一种是设备。对于设备，fcb_type 结构提供了 16 位的主/次设备号。不过，如前所述，在 devfs 中主设备号与具体的驱动程序没有固定的对应关系，而是动态分配的。每个 file_type 数据结构中都有个指针 ops，根据需要指向具体的 file_operations 或其他数据结构。

内核中还有个数据结构 fs_info，也定义于 fs/devfs/base.c：

```
668    struct fs_info       /*  This structure is for each mounted devfs  */
669    {
670        unsigned int num_inodes;    /*  Number of inodes created           */
671        unsigned int table_size;    /*  Size of the inode pointer table  */
672        struct devfs_entry **table;
673        struct super_block *sb;
674        volatile struct devfsd_buf_entry *devfsd_buffer;
675        volatile unsigned int devfsd_buf_in;
676        volatile unsigned int devfsd_buf_out;
677        volatile int devfsd_sleeping;
678        volatile int devfsd_buffer_in_use;
679        volatile struct task_struct *devfsd_task;
680        volatile struct file *devfsd_file;
681        volatile unsigned long devfsd_event_mask;
682        atomic_t devfsd_overrun_count;
683        wait_queue_head_t devfsd_wait_queue;
684        wait_queue_head_t revalidate_wait_queue;
685    };
```

这个数据结构中的指针 table 指向一个 devfs_entry 指针数组，table_size 为数组当前的大小。这个数组中的指针分别指向所有已经分配的 devfs_entry 数据结构，或者说 devfs 子树中当前的所有节点。一开始时内核中没有已经分配的 devfs_entry 数据结构，所以这个数组的大小为 0。

内核在设备驱动程序中仍然使用设备号，不过主/次设备号都已改成 16 位，并从中划出一块供 devfs 动态分配。原来已经分配了主/次设备号的还继续延用原来的设备号，而未经分配的则可以在初始化阶段向 devfs 登记时让 devfs 动态地分配一个。动态分配的设备号只是临时的，其作用仅在于在 devfs 中的节点与数组 chrdevs[ ]或 blk_dev[ ]中的元素建立起联系。这样，只要应用软件按预定的路径名打开设备文件，就能得到目标设备当前的设备号。

对于字符设备与块设备，供 devfs 动态分配的设备号都从 MIN_DEVNUM 开始（fs/devfs/base.c）：

```
527    #define MIN_DEVNUM 36864  /*  Use major numbers 144    */
528    #define MAX_DEVNUM 61439  /*  through 239, inclusive   */

688    static unsigned int next_devnum_char = MIN_DEVNUM;
689    static unsigned int next_devnum_block = MIN_DEVNUM;
```

第一个 devfs_entry 数据结构当然应该是 devfs 的根节点，函数 get_root_entry( )的代码在 fs/devfs/base.c 中：

[init_devfs_fs( ) > kern_mount( ) > read_super( ) > devfs_read_super( ) > get_root_entry( )]

```
841    /**
842     *  get_root_entry - Get the root devfs entry.
843     *
```

```
844      * Returns the root devfs entry on success, else %NULL.
845      */
846
847     static struct devfs_entry *get_root_entry (void)
848     {
849         struct devfs_entry *new;
850
851         /* Always ensure the root is created */
852         if (root_entry != NULL) return root_entry;
853         if ( ( root_entry = create_entry (NULL, NULL, 0) ) == NULL ) return NULL;
854         root_entry->registered = TRUE;
855         root_entry->mode = S_IFDIR;
856         /* Force an inode update, because lookup( ) is never done for the root */
857         update_devfs_inode_from_entry (root_entry);
858         /* And create the entry for ".devfsd" */
859         if ( ( new = create_entry (root_entry, ".devfsd", 0) ) == NULL )
860         return NULL;
861         new->registered = TRUE;
862         new->u.fcb.u.device.major = next_devnum_char >> 8;
863         new->u.fcb.u.device.minor = next_devnum_char & 0xff;
864         ++next_devnum_char;
865         new->mode = S_IFCHR | S_IRUSR | S_IWUSR;
866         new->u.fcb.default_uid = 0;
867         new->u.fcb.default_gid = 0;
868         new->u.fcb.ops = &devfsd_fops;
869         return root_entry;
870     }   /* End Function get_root_entry */
```

先创建 devfs 的根节点 root_entry，这里指针 root_entry 是个全局量。

[init_devfs_fs( ) > kern_mount( ) > read_super( ) > devfs_read_super( ) > get_root_entry( ) > create_entry( )]

```
764     static struct devfs_entry *create_entry (struct devfs_entry *parent,
765                          const char *name,unsigned int namelen)
766     {
767         struct devfs_entry *new, **table;
768
769         /* First ensure table size is enough */
770         if (fs_info.num_inodes >= fs_info.table_size)
771         {
772         if ( ( table = kmalloc (sizeof *table *
773                     (fs_info.table_size + INODE_TABLE_INC),
774                     GFP_KERNEL) ) == NULL ) return NULL;
775         fs_info.table_size += INODE_TABLE_INC;
776     #ifdef CONFIG_DEVFS_DEBUG
777         if (devfs_debug & DEBUG_I_CREATE)
778             printk ("%s: create_entry( ): grew inode table to: %u entries\n",
```

```
779                 DEVFS_NAME, fs_info.table_size);
780     #endif
781         if (fs_info.table)
782         {
783             memcpy (table, fs_info.table, sizeof *table *fs_info.num_inodes);
784             kfree (fs_info.table);
785         }
786         fs_info.table = table;
787         }
788         if ( name && (namelen < 1) ) namelen = strlen (name);
789         if ( ( new = kmalloc (sizeof *new + namelen, GFP_KERNEL) ) == NULL )
790         return NULL;
791         /*  Magic: this will set the ctime to zero, thus subsequent lookups will
792         trigger the call to <update_devfs inode_from_entry>  */
793         memset (new, 0, sizeof *new + namelen);
794         new->parent = parent;
795         if (name) memcpy (new->name, name, namelen);
796         new->namelen = namelen;
797         new->inode.ino = fs_info.num_inodes + FIRST_INODE;
798         new->inode.nlink = 1;
799         fs_info.table[fs_info.num inodes] = new;
800         ++fs_info.num_inodes;
801         if (parent == NULL) return new;
802         new->prev = parent->u.dir.last;
803         /*  Insert into the parent directory's list of children  */
804         if (parent->u.dir.first == NULL) parent->u.dir.first = new;
805         else parent->u.dir.last->next = new;
806         parent->u.dir.last = new;
807         return new;
808     }   /*  End Function create_entry  */
```

当 fs_info 数据结构中的 num_inodes 字段和 table_size 字段相等时，就需要扩充由 fs_info.table 所指的 devfs_entry 指针数组了。每次扩充都在原来的大小上增加 INODE_TABLE_INC，即 250 个指针的容量。显然，创建 devfs 的根节点时这两个字段都是 0，所以要为最初的 250 个指针分配空间。如果不是第一次分配空间，则还要把已经存在的数组复制到新的空间中，并释放原有的空间。接着，就要为 devfs_entry 数据结构本身(连同节点名字符串)分配空间。用于 devfs 根节点的 inode 号为 FIRST_INODE，即 1，以后则逐次递增。除根节点外，devfs 文件系统中的每个节点都有个父节点，而其父节点必定是个目录节点，802～806 行将新的节点与父节点以及同一目录中的其他节点链接在一起。

回到 get_root_entry( )中，下一步要在 devfs 的根节点下创建一个节点“.devfsd”。这个节点是个 fcb 节点。在 devfs 中，设备号是由系统自动分配的，设备号起点是（（（（？？？）））读者将看到，设备号在 devfs 中并不起着原来那么重要的作用。此外，fcb 节点都有个指针 ops 指向一个 file_operations 数据结构。对于节点“.devfsd”，这个数据结构是定义于 fs/devfs/base.c 的 devfsd_fops：

```
715     /*  Devfs daemon file operations  */
716     static struct file_operations devfsd_fops =
```

```
717     {
718         read:     devfsd_read,
719         ioctl:    devfsd_ioctl,
720         release:  devfsd_close,
721     };
```

这是个特殊的节点，这个节点并不是代表着一项具体的设备，而是反映着 devfs 当前的内容变化，其意图是在用户空间创建一个 devfs 的“保护神”进程 devfsd。这个进程的主体是个无穷循环，并总是因为读/dev/.devfsd 受阻而睡眠。但是，每当 devfs 的内容发生变化时，例如创建或删除一个节点或打开一个设备文件时，内核就(通过一个叫 devfsd_notify( )的函数)将这个进程唤醒。这样，就可以安排这个进程每当 devfs 的内容发生变化时就作出一些反应，例如在屏幕上显示一行信息“/dev/x/y/z/1 is up”。

回到 devfs_read_super( )的代码中，下面就是填写 devfs 的 super_block 数据结构的内容，注意这里把指针 s_op 设置成指向 devfs_sops，定义于 fs/devfs/base.c：

```
2364    static struct super_operations devfs_sops =
2365    {
2366        read_inode:     devfs_read_inode,
2367        write_inode:    devfs_write_inode,
2368        statfs:         devfs_statfs,
2369    };
```

然后就通过 get_vfs_inode( )为 devfs 的根节点创建一个 inode 数据结构(fs/devfs/base.c)：

[init_devfs_fs( ) > kern_mount( ) > read_super( ) > devfs_read_super( ) > get_vfs_inode ( )]

```
2381    static struct inode *get_vfs_inode (struct super_block *sb,
2382                                        struct devfs_entry *de,
2383                                        struct dentry *dentry)
2384    {
2385        struct inode *inode;
2386
2387        if (de->inode.dentry != NULL)
2388        {
2389        printk ("%s: get_vfs_inode(%u): \
                old de->inode.dentry: %p \"%s\"  new dentry: %p \"%s\"\n",
2390            DEVFS_NAME, de->inode.ino,
2391            de->inode.dentry, de->inode.dentry->d_name.name,
2392            dentry, dentry->d_name.name);
2393        printk ("  old inode: %p\n", de->inode.dentry->d_inode);
2394        return NULL;
2395        }
2396        if ( ( inode = iget (sb, de->inode.ino) ) == NULL ) return NULL;
2397        de->inode.dentry = dentry;
2398        return inode;
2399    }   /*  End Function get_vfs_inode  */
```

读者在第 5 章中已经看到，iget( )首先在 inode 结构的杂凑队列中查找，如果找不到就新创建一个。最后，通过 d_alloc_root( )为 devfs 的根节点创建一个 dentry 数据结构，这个节点也叫“/”，但只是 devfs 的根，与整个文件系统的总根不是一回事。

完成了对超级块的处理以后，kern_mount( )将 devfs 的根节点暂时“安装”到了 devfs 的 file_system_type 数据结构 devfs_fs_type 上，使其指针 kern_mnt 指向用作“连接件”的 vfsmount 结构，为以后进一步安装作好了准备。所以，这种初始的安装也可以称作“预安装”，有关详情可参阅/proc 文件系统的安装。

在安装了系统的总根以后，系统的初始化过程还会调用 mount_devfs_fs( )对 devfs 作进一步的安装(fs/devfs/base.c)：

```
3363    void __init mount_devfs_fs (void)
3364    {
3365        int err;
3366
3367        if ( (boot_options & OPTION_NOMOUNT) ) return;
3368        err = do_mount ("none", "/dev", "devfs", 0, "");
3369        if (err == 0) printk ("Mounted devfs on /dev\n");
3370        else printk ("Warning: unable to mount devfs, err: %d\n", err);
3371    }   /*  End Function mount_devfs_fs  */
```

可见，特殊文件系统 devfs 的安装点是“/dev”。由于 devfs_fs_type 中的 FS_SINGLE 标志位为 1，安装时会把保存在 devfs_fs_type 中的 vfsmount 结构指针安装到“/dev”节点上。

完成了 devfs 的安装以后，各种设备的驱动程序就可以通过 devfs_register_chrdev( )或 devfs_register_blkdev( )向 devfs 登记，从而在/dev 目录下创建起相应的节点。

(1) 通过 devfs_register_chrdev( )向 devfs 登记一类设备的主设备号、设备名以及 file_operations 数据结构，建立起三者间的联系。主设备号可以是静态指定的，也可以要求 devfs 动态地分配。

(2) 或者通过 devfs_register_blkdev( )向 devfs 登记一类设备的主设备号、设备名以及 block_device_operations 数据结构，建立起三者间的联系。同样，主设备号可以是静态指定的，也可以要求 devfs 动态度分配。

(3) 通过 devfs_mk_dir( )建立目录节点。

(4) 通过 devfs_register( )登记具体的设备，并在指定目录下建立叶节点。

在“可安装模块”一节所引的实例中，读者可以看到一个声卡驱动模块在初始化时首先通过 devfs_register_chrdev( )登记驱动程序，并且看到这个函数实际上通过 register_chrdev( )完成登记。所谓“登记”，实际上就是把给定的 file_operations 结构指针填入 chrdevs[ ]中的某个元素，而设备的主设备号，则就是用于这个数组的下标。在老的方案中，主设备号都是静态地指定好的，声卡设备的主设备号就是 SOUND_MAJOR。而在 devfs 中，则既可以继续延用静态的主设备号，也可以在调用 devfs_register_chrdev( )或 register_chrdev( )时以 0 为主设备号，表示要求 devfs 动态地分配一个。其实，不管是否采用 devfs，都要先进行登记，这样才能在打开文件时根据主设备号找到该设备的 file_operations 结构。不同的是，在老方案中这个主设备号必须是预知的，这样才能从内核外部通过 mknod( )在/dev 目录下创建起相应的节点。而在 devfs 中，则可以先动态分配主设备号(并登记)，然后

才在内核中(内部)通过 devfs_register( )在特殊文件系统中创建节点。而且，在 devfs_register( )之前还可以根据需要通过 devfs_mk_dir( )建立起若干中间节点。这样，只要在设备驱动模块与应用程序在路径名的使用上有默契就行了，主设备号的使用实际上是透明的。所以，devfs_mk_dir( )和 devfs_register( )才是 devfs 的关键所在。先看 devfs_mk_dir( )，其代码在 fs/devfs/base.c 中：

```
1530    /**
1531     *  devfs_mk_dir - Create a directory in the devfs namespace.
1532     *  @dir: The handle to the parent devfs directory entry. If this is %NULL the
1533     *      new name is relative to the root of the devfs.
1534     *  @name: The name of the entry.
1535     *  @info: An arbitrary pointer which will be associated with the entry.
1536     *
1537     *  Use of this function is optional. The devfs_register( ) function
1538     *  will automatically create intermediate directories as needed. This function
1539     *  is provided for efficiency reasons, as it provides a handle to a directory.
1540     *  Returns a handle which may later be used in a call to devfs_unregister( ).
1541     *  On failure %NULL is returned.
1542     */
1543
1544    devfs_handle_t devfs_mk_dir (devfs_handle_t dir, const char *name, void *info)
1545    {
1546        int is_new;
1547        struct devfs_entry *de;
1548
1549        if (name == NULL)
1550        {
1551        printk ("%s: devfs_mk_dir( ): NULL name pointer\n", DEVFS_NAME);
1552        return NULL;
1553        }
1554        de = search_for_entry (dir, name, strlen (name), TRUE, TRUE, &is_new,
1555                   FALSE);
1556        if (de == NULL)
1557        {
1558        printk ("%s: devfs_mk_dir( ): could not create entry: \"%s\"\n",
1559            DEVFS_NAME, name);
1560        return NULL;
1561        }
1562        if (!S_ISDIR (de->mode) && de->registered)
1563        {
1564        printk ("%s: devfs_mk_dir( ): existing non-directory entry: \"%s\"\n",
1565            DEVFS_NAME, name);
1566        return NULL;
1567        }
1568    #ifdef CONFIG_DEVFS_DEBUG
1569        if (devfs_debug & DEBUG_REGISTER)
1570        printk ("%s: devfs_mk_dir(%s): de: %p %s\n",
1571            DEVFS_NAME, name, de, is_new ? "new" : "existing");
```

```
1572    #endif
1573        if (!S_ISDIR (de->mode) && !is_new)
1574        {
1575        /*  Transmogrifying an old entry  */
1576        de->u.dir.first = NULL;
1577        de->u.dir.last = NULL;
1578        }
1579        de->mode = S_IFDIR | S_IRUGO | S_IXUGO;
1580        de->info = info;
1581        if (!de->registered) de->u.dir.num_removable = 0;
1582        de->registered = TRUE;
1583        de->show_unreg = (boot_options & OPTION_SHOW) ? TRUE : FALSE;
1584        de->hide = FALSE;
1585        return de;
1586    }   /*  End Function devfs_mk_dir  */
```

参数 dir 指向父目录的 devfs_entry 结构，数据类型 devfs_handle_t 实际上就是 devfs_entry 指针，定义于 include/linux/devfs_fs_kernel.h：

```
  45    typedef struct devfs_entry * devfs_handle_t;
```

这个函数的主体是 search_for_entry( )，通过它在 dir 下面的子树中找到或者创建目标节点的 devfs_entry 结构。然后，如果是新创的节点，则加以设置。函数 search_for_entry( )的代码也在 fs/devfs/base.c 中：

[devfs_mk_dir( ) > search_for_entry( )]

```
 873    /**
 874     *  search_for_entry - Search for an entry in the devfs tree.
 875     *  @dir: The parent directory to search from. If this is %NULL the root is used
 876     *  @name: The name of the entry.
 877     *  @namelen: The number of characters in @name.
 878     *  @mkdir: If %TRUE intermediate directories are created as needed.
 879     *  @mkfile: If %TRUE the file entry is created if it doesn't exist.
 880     *  @is_new: If the returned entry was newly made, %TRUE is written here. If
 881     *      this is %NULL nothing is written here.
 882     *  @traverse_symlink: If %TRUE then symbolic links are traversed.
 883     *
 884     *  If the entry is created, then it will be in the unregistered state.
 885     *  Returns a pointer to the entry on success, else %NULL.
 886     */
 887
 888    static struct devfs_entry *search_for_entry (struct devfs_entry *dir,
 889                                 const char *name,
 890                                 unsigned int namelen, int mkdir,
 891                                 int mkfile, int *is_new,
 892                                 int traverse_symlink)
```

```
893     {
894         int len;
895         const char *subname, *stop, *ptr;
896         struct devfs_entry *entry;
897
898         if (is_new) *is_new = FALSE;
899         if (dir == NULL) dir = get_root_entry( );
900         if (dir == NULL) return NULL;
901         /*  Extract one filename component  */
902         subname = name;
903         stop = name + namelen;
904         while (subname < stop)
905         {
906         /*  Search for a possible '/'  */
907         for (ptr = subname; (ptr < stop) && (*ptr != '/'); ++ptr);
908         if (ptr >= stop)
909         {
910             /*  Look for trailing component  */
911             len = stop - subname;
912             entry = search_for_entry_in_dir (dir, subname, len,
913                                 traverse_symlink);
914             if (entry != NULL) return entry;
915             if (!mkfile) return NULL;
916             entry = create_entry (dir, subname, len);
917             if (entry && is_new) *is_new = TRUE;
918             return entry;
919         }
920         /*  Found '/': search for directory  */
921         if (strncmp (subname, "../", 3) == 0)
922         {
923             /*  Going up  */
924             dir = dir->parent;
925             if (dir == NULL) return NULL;  /*  Cannot escape from devfs  */
926             subname += 3;
927             continue;
928         }
929         len = ptr - subname;
930         entry = search_for_entry_in_dir (dir, subname, len, traverse_symlink);
931         if (!entry && !mkdir) return NULL;
932         if (entry == NULL)
933         {
934             /*  Make it  */
935             if ( ( entry = create_entry (dir, subname, len) ) == NULL )
936             return NULL;
937             entry->mode = S_IFDIR | S_IRUGO | S_IXUGO | S_IWUSR;
938             if (is_new) *is_new = TRUE;
939         }
940         if ( !S_ISDIR (entry->mode) )
```

```
941             {
942                 printk ("%s: existing non-directory entry\n", DEVFS_NAME);
943                 return NULL;
944             }
945             /*  Ensure an unregistered entry is re-registered and visible  */
946             entry->registered = TRUE;
947             entry->hide = FALSE;
948             subname = ptr + 1;
949             dir = entry;
950             }
951             return NULL;
952     }   /*  End Function search_for entry  */
```

这个函数返回指向找到或新创建的 devfs_entry 结构的指针，并通过副作用返回参数 is_new，表明这个 devfs_entry 结构是否新创。如果参数 dir 为 0，则以 devfs 的根节点为父节点。参数 name 指向一个相对路径名。代码中通过一个 while 循环顺着相对路径名中的节点逐步向前推进，每一步都通过 search_for_entry_in_dir( )在当前目录中找到下一个节点(930 行)，如果找不到就通过 create_entry( )自动补上一个中间节点。最后，当路径名中剩下的部分不再有“/”字符存在时，这就是目标节点了(908 行)。对于目标节点，也是先通过 search_for_entry_in_dir( )在当前目录中寻找，如果找不到就通过 create_entry 创建。对于读过 path_walk( )的读者，这只不过是“小菜一碟”。至于 search_for_entry_in_dir( )，其代码也在 fs/devfs/base.c 中，我们把它列在下面供读者自己阅读，从中可以理解对 devfs_entry 结构中的 union 的运用。

[devfs_mk_dir( ) > search_for_entry( ) > search_for_entry_in_dir( )]

```
738     static struct devfs_entry *search_for_entry_in_dir (struct devfs_entry *parent,
739                                   const char *name,
740                                   unsigned int namelen,
741                                   int traverse_symlink)
742     {
743         struct devfs_entry *curr;
744 
745         if ( !S_ISDIR (parent->mode) )
746         {
747         printk ("%s: entry is not a directory\n", DEVFS_NAME);
748         return NULL;
749         }
750         for (curr = parent->u.dir.first; curr != NULL; curr = curr->next)
751         {
752         if (curr->namelen != namelen) continue;
753         if (memcmp (curr->name, name, namelen) == 0) break;
754         /*  Not found: try the next one  */
755         }
756         if (curr == NULL) return NULL;
757         if (!S_ISLNK (curr->mode) || !traverse symlink) return curr;
```

```
758         /*  Need to follow the link: this is a stack chomper  */
759         return search_for_entry (parent,
760                      curr->u.symlink.linkname, curr->u.symlink.length,
761                      FALSE, FALSE, NULL, TRUE);
762     }   /*  End Function search_for_entry_in_dir  */
```

通过 devfs_mk_dir( )创建了所需的目录节点以后，就在 devfs 的目录系统中为目标设备建立起了一个框架，最后还要通过 devfs_register( )为具体的设备创建目标节点。这个函数的代码在 fs/devfs/base.c 中：

```
1214    /**
1215     *  devfs_register - Register a device entry.
1216     *  @dir: The handle to the parent devfs directory entry. If this is %NULL the
1217     *      new name is relative to the root of the devfs.
1218     *  @name: The name of the entry.
1219     *  @flags: A set of bitwise-ORed flags (DEVFS_FL_*).
1220     *  @major: The major number. Not needed for regular files.
1221     *  @minor: The minor number. Not needed for regular files.
1222     *  @mode: The default file mode.
1223     *  @ops: The &file_operations or &block_device operations structure.
1224     *      This must not be externally deallocated.
1225     *  @info: An arbitrary pointer which will be written to the @private_data
1226     *  field of the &file structure passed to the device driver. You can set
1227     *  this to whatever you like, and change it once the file is opened
1228     *  (the next file opened will not see this change).
1229     *
1230     *  Returns a handle which may later be used in a call to devfs_unregister( ).
1231     *  On failure %NULL is returned.
1232     */
1233
1234    devfs_handle_t devfs_register (devfs_handle_t dir, const char *name,
1235                       unsigned int flags,
1236                       unsigned int major, unsigned int minor,
1237                       umode_t mode, void *ops, void *info)
1238    {
1239        int is_new;
1240        struct devfs_entry *de;
1241
1242        if (name == NULL)
1243        {
1244        printk ("%s: devfs_register( ): NULL name pointer\n", DEVFS_NAME);
1245        return NULL;
1246        }
1247        if (ops == NULL)
1248        {
1249        if ( S_ISBLK (mode) ) ops = (void *) get_blkfops (major);
1250        if (ops == NULL)
```

```
1251            {
1252                printk ("%s: devfs_register(%s): NULL ops pointer\n",
1253                    DEVFS_NAME, name);
1254                return NULL;
1255            }
1256            printk ("%s: devfs_register(%s): NULL ops, got %p from major table\n",
1257                DEVFS_NAME, name, ops);
1258            }
1259            if ( S_ISDIR (mode) )
1260            {
1261            printk("%s: devfs_register(%s): creating directories is not allowed\n",
1262                    DEVFS_NAME, name);
1263            return NULL;
1264            }
1265            if ( S_ISLNK (mode) )
1266            {
1267            printk ("%s: devfs_register(%s): creating symlinks is not allowed\n",
1268                DEVFS_NAME, name);
1269            return NULL;
1270            }
1271            if ( S_ISCHR (mode) && (flags & DEVFS_FL_AUTO_DEVNUM) )
1272            {
1273            if (next_devnum_char >= MAX_DEVNUM)
1274            {
1275                printk ("%s: devfs_register(%s): exhausted char device numbers\n",
1276                    DEVFS_NAME, name);
1277                return NULL;
1278            }
1279            major = next_devnum_char >> 8;
1280            minor = next_devnum_char & 0xff;
1281            ++next_devnum_char;
1282            }
1283            if ( S_ISBLK (mode) && (flags & DEVFS_FL_AUTO_DEVNUM) )
1284            {
1285            if (next_devnum_block >= MAX_DEVNUM)
1286            {
1287                printk ("%s: devfs_register(%s): exhausted block device numbers\n",
1288                    DEVFS_NAME, name);
1289                return NULL;
1290            }
1291            major = next_devnum_block >> 8;
1292            minor = next_devnum_block & 0xff;
1293            ++next_devnum_block;
1294            }
1295            de = search_for_entry (dir, name, strlen (name), TRUE, TRUE, &is_new,
1296                        FALSE);
1297            if (de == NULL)
1298            {
```

```
1299        printk ("%s: devfs_register( ): could not create entry: \"%s\"\n",
1300            DEVFS_NAME, name);
1301        return NULL;
1302        }
1303    #ifdef CONFIG_DEVFS_DEBUG
1304        if (devfs_debug & DEBUG_REGISTER)
1305        printk ("%s: devfs_register(%s): de: %p %s\n",
1306            DEVFS_NAME, name, de, is_new ? "new" : "existing");
1307    #endif
1308        if (!is_new)
1309        {
1310        /*  Existing entry  */
1311        if ( !S_ISCHR (de->mode) && !S_ISBLK (de->mode) &&
1312             !S_ISREG (de->mode) )
1313        {
1314            printk ("%s: devfs register( ): existing non-device/file entry: \"%s\"\n",
1315                DEVFS NAME, name);
1316            return NULL;
1317        }
1318        if (de->registered)
1319        {
1320            printk("%s: devfs_register( ): device already registered: \"%s\"\n",
1321               DEVFS_NAME, name);
1322            return NULL;
1323        }
1324        }
1325        de->registered = TRUE;
1326        if ( S_ISCHR (mode) || S_ISBLK (mode) )
1327        {
1328        de->u.fcb.u.device.major = major;
1329        de->u.fcb.u.device.minor = minor;
1330        }
1331        else if ( S_ISREG (mode) ) de->u.fcb.u.file.size = 0;
1332        else
1333        {
1334        printk ("%s: devfs_register( ): illegal mode: %x\n",
1335            DEVFS_NAME, mode);
1336        return (NULL);
1337        }
1338        de->info = info;
1339        de->mode = mode;
1340        if (flags & DEVFS_FL_CURRENT_OWNER)
1341        {
1342        de->u.fcb.default_uid = current->uid;
1343        de->u.fcb.default_gid = current->gid;
1344        }
1345        else
1346        {
```

```
1347            de->u.fcb.default_uid = 0;
1348            de->u.fcb.default_gid = 0;
1349            }
1350            de->registered = TRUE;
1351            de->u.fcb.ops = ops;
1352            de->u.fcb.auto_owner = (flags & DEVFS_FL_AUTO_OWNER) ? TRUE : FALSE;
1353            de->u.fcb.aopen_notify = (flags & DEVFS_FL_AOPEN_NOTIFY) ? TRUE : FALSE;
1354            if (flags & DEVFS_FL_REMOVABLE)
1355            {
1356            de->u.fcb.removable = TRUE;
1357            ++de->parent->u.dir.num_removable;
1358            }
1359            de->u.fcb.open = FALSE;
1360            de->show_unreg = ( (boot_options & OPTION_SHOW)
1361                    || (flags & DEVFS_FL_SHOW_UNREG) ) ? TRUE : FALSE;
1362            de->hide = (flags & DEVFS_FL_HIDE) ? TRUE : FALSE;
1363            de->no_persistence = (flags & DEVFS_FL_NO_PERSISTENCE) ? TRUE : FALSE;
1364            devfsd_notify (de, DEVFSD_NOTIFY_REGISTERED, flags & DEVFS_FL_WAIT);
1365            return de;
1366        }   /*  End Function devfs_register  */
```

我们知道，“设备文件”节点的实质就在于：把代表着目标设备的文件名(路径名)与具体的“对象”挂上钩，包括一套驱动程序（通过主设备号）以及具体的数据结构(通过次设备号)。在老的方案中，主设备号就惟一地确定了一个 file_operations 数据结构。相比之下，在 devfs 中提供了更大的灵活性，在调用 devfs_register( )时可以通过参数 ops 传下一个 file_operations 结构指针，只是当 ops 为 0 时才采用由主设备号确定的 file_operations 数据结构。主/次设备号则既可以作为参数传下来，也可以由系统产生。

同样，这里也是通过 search_for_entry( )在以 dir 为根的子树中找到或创建目标节点的 devfs_entry 数据结构(1295 行)，只不过这一次把 devfs_entry 结构用作 fcb 节点，而不是目录节点。最后，这里还调用了一个函数 devfsd_notify( )，目的在于为 devfsd 进程准备下一些信息并将其唤醒，让它知道现在 devfs 中又多了一个设备节点。

此后应用进程就可以通过新的 devfs 格式的路径名打开设备文件，此后的操作就与原来的没有什么不同了。显然，如果把目标节点直接建立在 devfs 的根目录，即/dev 下面，并且符合从前对设备名的约定，那么从应用程序的角度看就与老的方案没有什么不同了。

第9章

多处理器 SMP 系统结构

9.1 概述

在过去的10年里，微处理器的速度增长了近100倍，但是尽管如此，CPU的最高速度总是受到当时技术条件的限制。在给定的时间里，CPU能达到的最高速度总是给定的，再要提高速度就只好设法增加并行度，方法之一就是在计算机中使用多个处理器，所以，多处理器结构很长时间以来一直是计算机科学与技术的一个重要分支。在长期的研究中，人们先后开发了几种重要的多处理器系统结构模型，其中之一就是所谓“对称多处理器结构”(Semetric Multi-Processor Architecture)，缩写为SMP，这是一种比较简单的多处理器系统结构。SMP是一种系统结构的模型，不过为行文的方便我们在本书中也常常把采用SMP结构的计算机系统简称为SMP结构。还有，在本书中讲到SMP结构时大多专指采用Intel Pentium处理器的系统，但是也有些场合是泛指，读者要根据上下文加以区分。

在这种系统结构中，所有的CPU在运行时（除系统引导和初始化以外）都是“对称”的，或者说都是“平等”的，没有主次之分，通常物理上也采用同一种CPU（在这样的系统中，实际上已经没有所谓“中央处理器”了，但为了行文方便我们仍使用CPU表示“处理器”)。所有的CPU通过同一条总线共享同一个内存以及所有的外设。为了减少访问内存的冲突，SMP结构中的各个CPU通常都有自己的高速缓存。图9.1给出了SMP系统结构模型示意图。

对于Intel的Pentium处理器（以及高于Pentium的处理器，下同)，系统中最多可以容纳32个平行的处理器。

各个CPU动态地从系统的就绪进程队列中调度（挑选出）进程加以执行。一个进程在不同的时间中可以在不同的CPU上运行；中断请求则动态地分配给其中的某个CPU，由这个CPU提供中断服务。除一般的共享内存外，处理器间的通信手段还有进程间通信和处理器之间的中断请求。如前所述，通常每个CPU都配备了自己的高速缓冲存储器，以减少访问内存时的冲突；另一方面，虽然系统中所有的处理器都采用相同的时钟脉冲，但是由于指令的长短不一，加上访问内存时可能会有的冲突以及高速缓存的使用等等因素，一般而言指令的边界是不能对齐的。因此，系统中的各个CPU都在独立地、异步地执行指令。

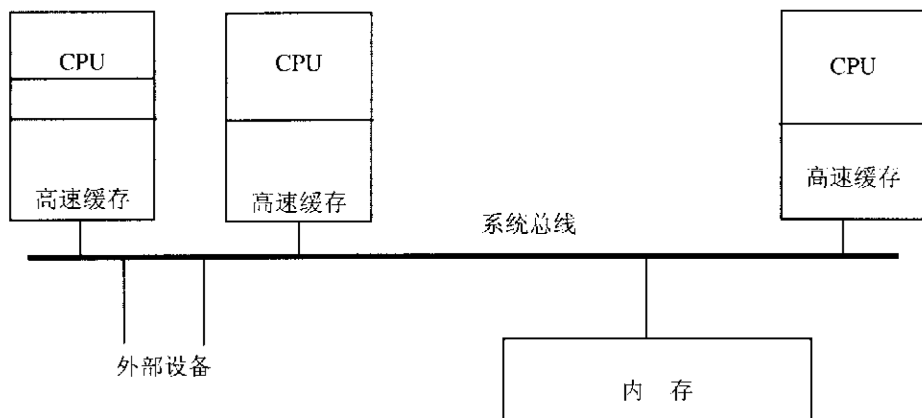


图 9.1 SMP 结构模型

这样的结构适合于要求提高系统总的“吞吐量”，但是系统中的各个进程则互相独立的那些应用。例如，网络服务器就是一个很好的例子，因为网络服务器在同一时间中要响应大量的“点击”，而对这些点击的服务又是互相独立的（尽管实际执行的程序很可能相同）。考虑到这一点，以及近年来对网络服务器的大量需求，就不难明白为什么这几年 SMP 结构变得这么热门了。至于是否可以通过采用 SMP 结构的计算机来提高解算大问题的速度，则取决于是否存在好的并行算法，从而可以有效地把给定的大问题分解成若干可以并发执行的进程。另一方面，SMP 是一种通用的系统结构，用它来实现一些重要的并行算法往往不如一些专用的系统结构（如“向量机”结构）那么有效。

但是，与单处理器结构相比，SMP 结构的实现有一些特殊的问题需要考虑和解决。

首先是处理器间的同步与互斥。从宏观上说这也是个进程间通信的问题，但是多个处理器的存在使这个问题更复杂化了。在单处理器结构中，各个进程在宏观上是并行的，但是在微观上却是串行的，因为在同一时间点上只有一个进程真正在运行（系统中只有一个处理器），因此称为“并发”。在这样的系统中，保证进程间的同步与互斥是比较容易的。回顾一下“临界区”的实现，就可以明白进程间的同步实际上可以归结到对临界资源的互斥操作。在单处理器结构中，只要能保证在对临界资源的操作中途不会发生进程调度，并且不会发生中断，或者即使发生了中断也与操作的对象无关，就保证了操作的互斥性。即使在极端的情况下（例如不允许关中断），只要对临界资源的操作能在单条指令中完成，那也保证了操作的互斥性，因为中断只能发生于指令之间，而不会发生在执行一条指令的中途。一般而言，只要能保证对临界资源操作的“原子性”，互斥性就保证了。所以，在单处理器结构中，能够在单条指令中完成的操作就认为是“原子操作”。这也是为什么一些 CPU 指令系统中设置了“测试并设置”、“测试并清除”等指令的原因，这些指令主要都是用于对临界资源的互斥操作。可是，在 SMP 结构中就不同了，由于系统中有多个处理器在独立地运行，即使能在单条指令中完成的操作也有可能受到干扰。就以“测试并设置”这条指令为例，它先从某个内存单元读出其内容，测试其中的某一位，并把这一位设置成 1（也可能原来就是 1），再写回内存单元中，并根据测试的结果设置标志位寄存器中的相应标志位。可见，在这条指令的执行过程中要访问内存两次，形成一个“读—改—写”的过程。这个过程中的每一步都是一个“微操作”，整条指令则由若干微操作构成。于是，在 SMP 结构中就完全有可能发生这样的情况：

(1) CPU1 从一个特定的内存单元读出，并测试到其中的某一位(假定是 bit4)为 0，然后将这一位

改成 1。

(2) 在 CPU1 还没有来得及把修改后的结果写回该内存单元之前，CPU2 也从同一内存单元读出，也测试到 bit4 为 0，然后也将这一位改成 1。

(3) 然后，CPU1 把修改后的结果写回内存单元。由于该内存单元的 bit4 原来是 0，CPU1 便以为取得了该项临界资源，并且确信不会有其他进程前来打扰。在 CPU1 看来，它已经把这个内存单元的 bit4 改成了 1，如果其他进程再来对此项临界资源执行“测试并设置”操作，就会因此而被关在门外。

(4) 然后，CPU2 也把修改后的结果写回内存单元，也以为取得了该项临界资源，也确信不会有其他进程可以前来打扰。

可是，实际上这两个 CPU 之间的互相干扰已是十分可能的了。这个例子告诉我们，与单处理器结构相比，SMP 结构对互斥操作的微观“分辨率”需更高，有些在单处理器结构中的“原子操作”在 SMP 结构中不再是原子的了。

第二个问题是高速缓存与内存之间(内容的)的一致性问题。在单处理结构中，使用高速缓存的目的仅在于通过提高 CPU 取指令和读/写数据的速度来改善系统的性能。高速缓存与内存的关系跟缓冲页面与磁盘上记录块的关系相似。当 CPU 要从内存读数据时，高速缓存的硬件会先根据目标地址检查这部分数据是否已经在高速缓存中，如果是，就不需要从内存中读了；否则，就从内存将目标所在的一小块数据，称为一条缓冲线(cache line)，以较高的速度装入高速缓存。一旦装入了高速缓存，再要从属于同一缓冲线的内存单元读数据时就能在高速缓存中“命中”，因而不需要到内存中去读了。高速缓存中的内容也像缓冲页面那样随时间而“老化”，当高速缓存的容量不够时就把最老的缓冲线内容丢弃，从而达到周转使用高速缓存空间的目的。此外，软件也可以通过特定的指令主动将高速缓存中属于某个内存区间的内容丢弃，以后要从这个区间读时就又从内存装入。读者以前看到过，通过 DMA 操作，从外设读入以后要丢弃与 DMA 缓冲区对应的缓冲线内容，就是因为 DMA 读操作改变了内存中 DMA 缓冲区的内容，使得高速缓存与内存不一致了。DMA 操作是由设备驱动程序安排启动的，所以设备驱动程序知道应该在操作完成以后丢弃高速缓存的内容。但是，在 SMP 结构中情况就更复杂了，因为一个 CPU 并不知道别的 CPU 会在何时改变内存的内容。采用高速缓存时的写操作有两种模式，一种称为“穿透”(Write-Through)模式，在这种模式中高速缓存对于写操作就好像不存在一样，每次写时都直接写到内存中，所以实际上只是对读操作使用高速缓存，因而效率相对较低。另一种称为“回写”(Write-Back) 模式，写的时候先写入高速缓存，然后由高速缓存的硬件在周转使用缓冲线时自动写入内存，或者由软件主动地“冲刷”有关的缓冲线。因此，在改变了缓冲页面的内容，并启动 DMA 写操作将其写回磁盘之前要先“冲刷”高速缓存中有关的缓冲线，因为改变了的内容可能还没有回写到内存缓冲区中。

从功能和作用的角度看，可以把高速缓存分成三部分。

(1) 对访问内存的缓冲(Cache)，其作用类似于访问磁盘时的缓冲，包括对数据和指令两方面的缓冲，这一部分是本来意义上的“高速缓存”。不过，对指令的缓冲基本上是透明的，我们这里关心的只是对数据的缓冲。此外，高速缓存在物理上往往分成一级(L1)和二级(L2)两部分，但是从软件的角度看并无不同。

(2) 页面映射目录和页面映射表的缓冲存储(TLB)。我们在第 2 章中曾经讲到，为了加快页面映射的速度，在页面映射的过程实际上并不是每次都到内存中访问页面映射目录和页面映射表，而是将它们所在的页面装入 CPU 内部，以加快地址转换的速度。这里所谓“CPU 内部”

其实就是高速缓存的一部分，称为 TLB，即“地址转换/查找缓冲区”（Translation Lookaside Buffers），因为其目的在于地址映射。

(3) 第三部分是写内存缓冲区(Write Buffer)。在多处理器系统中，当一个处理器启动对内存的写操作时，有可能系统总线已被锁住，或者其他处理器正在装入/写回一条缓冲线，此时 CPU 把要写的内容暂时放在缓冲区中，继续往下执行而不必停下来等待。然后，当条件允许时，缓冲区的有关硬件会把暂时缓冲着的内容写回内存。

高速缓存与内存一致性的问题，对于 TLB 和数据的缓冲在表现上和处理上有所不同。

在 CPU 中有个寄存器，称为“存储类型及范围寄存器”(Memory Type Range Register)MTRR，通过这个寄存器可以将内存中的不同区间设置成使用或不使用高速缓存，以及对于写操作采用穿透模式或回写模式。具体地，可以把每一个内存区间设置成下列几种模式：

(1) 不缓冲。

(2) 缓冲，穿透模式。

(3) 缓冲，回写模式。

(4) 写保护，只允许读不允许写。

此外，MTRR 还支持一种“结合写”(write combining)模式，用于跟写入次序无关的区间。例如，图像缓冲区就是这样的区间：把一个像素很快地先写成红，再写成黄，再写成绿，跟直接写成绿在效果上并无不同。当然，这种模式对于操作系统内核是不适合的。

其实，除 MTRR 以外，通过其他控制寄存器也能从总体上或按页面方式控制高速缓存的方式，只是没有采用 MTRR 时那样的灵活性和控制精度而已。再说在 Pentium 以前的处理器中根本就没有 MTRR，所以在 Linux 内核中是否采用 MTRR 是个选择项。

还有个与高速缓存密切相关的重要问题，就是高速缓冲的运用有可能改变对内存操作的次序。假定有两个观察者，一个观察 CPU 内部高速缓存受到访问的次序，而另一个观察内存受到访问的次序，则二者可能会有相当大的差异。前者就是程序中编排好的次序，所以称作“指令序”(program ordering)。后者则是实际出现在处理器外部，即系统总线上的次序，所以称作“处理器序”(processor ordering)。显然，不使用高速缓存时二者必然相同。而如果使用高速缓冲，那就要看具体的情况和操作。如果保证“处理器序”与“指令序”相同，就称作“强序”(strong ordering)；反之，如果“处理器序”有时候可能不同于“指令序”，就称作“弱序”(weak ordering)。对于单处理器结构的系统，这二者的不同并不成什么问题，然而对 SMP 结构的系统却可能成为问题。

此外，gcc 在编译过程中进行的优化也可能会改变操作的次序。这种情况下的“指令序”本身就可能与程序编写者的意图不同，在单处理器系统中已可能造成问题，在 SMP 结构中就更加可能了。

在 SMP 结构中，高速缓存的作用比在单处理器结构中更为重要，因为它不但可以提高取指令和读/写数据的速度，还有利于减少多个 CPU 在访问内存时的冲突。一般的内存都不允许在同一时间内（同一个“内存访问周期”内）受到多个 CPU 的访问，所以，在 SMP 结构中通常每个 CPU 都有较大的高速缓存，从而一旦把高速缓存装满以后，就可以运行相当长的时间而无需经常地读/写物理上的内存。

如上所述，单处理器系统中的 DMA 操作都是由设备驱动程序主动地启动的，所以设备驱动程序（确切地说是程序员）知道什么时候应该丢弃哪些缓冲线的内容，什么时候应该冲刷哪些缓冲线的内容。可是在 SMP 结构中就复杂了，此时每个 CPU 都有可能改变内存中的内容，而且是异步地改变。就是说，每个 CPU 都只知道自己何时会改变内存的内容(包括自己启动的 DMA 操作)，但是都不知道别的 CPU 会在什么时候改变内存的内容，也不知道自己本地的高速缓存中的内容是否已经与内存中不

一致。反过来，每个 CPU 都可能因为改变了内存的内容而使其他 CPU 的高速缓存变得不一致了。

第三个问题是对中断的处理。在单处理器结构中，整个系统只有一个 CPU，所有的中断请求都由这个 CPU 响应和处理。可是，在 SMP 结构中怎么办呢？是固定让其中的某一个 CPU 处理所有的中断请求，还是让所有的 CPU 轮流处理？如果是固定让一个 CPU 处理，那么其他的 CPU 是否就没有任何中断，连称为“心跳”的时钟中断也没有了？要是那样，万一在那些 CPU 上运行的进程陷入了死循环，从而永远没有机会进行系统调用，这些 CPU 岂不是就永远不会有进程调度了？如果是让所有的 CPU 轮流处理，或者谁空闲谁处理，那又怎样把中断请求分配给不同的 CPU 呢？

还有，后面读者会看到，为了解决上述的问题，就像进程之间要有“进程间通信”的手段一样，处理器之间也要有“处理器间中断”的手段，这显然不是单靠软件就能解决的。

下面，我们将结合 Intel 的 i386 结构，主要是 Pentium 处理器，以及 Linux 内核中的有关源代码来看这些问题是如何解决的。这里要说明一点，限于本书的篇幅，我们不可能对 SMP 结构这个课题作全面而详尽的介绍，那本身就已经足够一本专著的内容。事实上，Curt Schimmel 的《UNIX Systems for Modern Architectures》一书就是这方面的经典著作。同时，我们也不可能就 Pentium 处理器对 SMP 结构的支持作详尽的介绍。需要深入研究 SMP 结构的读者应该进一步阅读有关的专著和 Intel 提供的技术资料（可从 Intel 的网站下载）。

9.2　SMP 结构中的互斥问题

Intel 在 80386 的设计时就在一定程度上考虑到了 SMP 结构的需要。后来，随着应用中逐渐增长的需求和技术的发展，又对 i386 系统结构逐步作了加强，到推出 Pentium 处理器（更确切地说是 P6 系列）的时候就已经有了比较完整的解决方案。

先看 i386 系列的处理器，特别是 Pentium，是怎样解决 SMP 结构中对临界资源操作的原子性问题的。首先，单纯的读或写本来就是原子的。不管是 8 位、16 位，还是 32 位的读或写，都只要一个内存读/写微操作就能完成，所以是不可分割的。问题在于一些既要读又要写，因而需要两个或两个以上的微操作才能完成的指令，如前述的“读—改—写”操作就是一个例子。对于这样的指令，i386 CPU 提供了在指令执行期间对总线加锁的手段。CPU 芯片上有一条引线 LOCK，如果汇编语言的程序中在一条指令前面加上前缀“LOCK”，经过汇编以后的机器代码就使 CPU 在执行这条指令的时候把引线 LOCK 的电位拉低，从而把总线锁住，这样同一总线上别的 CPU 就暂时不能通过总线访问内存了。这方面还有个特例，就是在执行指令 xchg 的时候 CPU 会自动将总线锁住，而不需要在程序中使用前缀“LOCK”。xchg 指令将一个内存单元中的内容与一个寄存器的内容对换，因此常常用于对内核信号量(Semephore)的操作。显然，这是一条既有读又有写的操作指令。下面的函数 spin_trylock( )就是一个使用 xchg 的实例，取自 include/asm-i386/spinlock.h：

```
68	static inline int spin_trylock(spinlock_t *lock)
69	{
70		char oldval;
71		__asm__ __volatile__(
72			"xchgb %b0,%1"
```

```
73              :"=q" (oldval), "=m" (lock->lock)
74              :"0" (0) : "memory");
75          return oldval > 0;
76      }
```

这个函数将操作数%0，即 oldval 先设置成 0，然后与操作数%1，即内存单元 lock->lock 的内容对换。如果 lock->lock 的内容原来就是 0，就说明这个锁 lock 在此之前已经被锁上了，所以 spin_trylock( )返回 0 表示加锁失败。反之，如果 lock->lock 的内容原来非 0，则现在变成了 0，加锁就成功了，所以 spin_trylock( )返回 1。由于在执行 xchg 指令时 CPU 会自动锁住总线，所以不需要在这条指令前面加前缀“LOCK”。再看对这个函数的使用(kernel/softirq.c)：

```
244     spinlock_t global_bh_lock = SPIN_LOCK_UNLOCKED;
245
246     static void bh_action(unsigned long nr)
247     {
248         int cpu = smp_processor_id( );
249
250         if (!spin_trylock(&global_bh_lock))
251             goto resched;
        ......
256         if (bh_base[nr])
257             bh_base[nr]( );
        ......
260         spin_unlock(&global_bh_lock);
261         return;
        ......
265     resched:
266         mark_bh(nr);
267     }
```

这里的 global_bh_lock 是用于 bh 函数执行机制的锁，spinlock_t 数据结构的定义在 include/asm-i386/spinlock.h 中：

```
17      /*
18       * Your basic SMP spinlocks, allowing only a single CPU anywhere
19       */
20
21      typedef struct {
22          volatile unsigned int lock;
23      #if SPINLOCK_DEBUG
24          unsigned magic;
25      #endif
26      } spinlock_t;
```

从 250 行的 spin_trylock( )至 260 行的 spin_unlock( )之间是需要保证互斥、且在同一时间中只允许一个 CPU 在里面执行的临界区。所以，CPU 在进入这个区间前，先要通过 spin_trylock( )加锁，而

global_bh_lock 的初始值非 0。如果 spin_trylock( )的返回值为 0 就表示已经有 CPU 在里面执行了，所以就跳过对 bh 函数的执行而转到 resched 处。这样，利用 xchg 指令的原子性，就实现了对临界资源 global_bh_lock 操作的互斥性，进而实现了整个临界区中操作的互斥性。

显然，在竞争(或者说抢夺)临界资源时，只有系统中所有的 CPU 都确实是在对同一个内存单元执行 xchg 指令才有意义。如果各个 CPU 只是对 global_bh_lock 在其高速缓存中的一个副本执行 xchg 指令，而执行的结果又不能立即为其他 CPU 所见，则 spin_trylock( )就毫无作用了。读者在下一节中将看到，Intel 在 Pentium 处理器中提供了一种称为 snooping 的机制，能自动维持高速缓存与内存的数据一致性，使这个问题对于软件成为透明。否则，就必须把 global_bh_lock 放在一个不加缓冲的区间中。

再来看（计数）信号量的实现，函数 down( )的代码读者以前看到过了，这里只列出其中几行供读者比较和领会(include/asm-i386/semaphore.h)：

```
114    static inline void down(struct semaphore * sem)
115    {
       . . . . . .
120            asm __volatile__(
121              "# atomic down operation\n\t"
122              LOCK "decl %0\n\t"      /* --sem->count */
123              "js 2f\n"
       . . . . . .
129              :"=m" (sem->count)
130              :"c" (sem)
131              :"memory");
132    }
```

这里使 sem->count 的数值减 1 的操作是通过 dec 指令完成的。显然，这条指令执行的也是“读—改—写”操作，为了保证操作的原子性，在指令前面加上了前缀“LOCK”。

前面讲过，高速缓存的使用可能会使实际的内存操作改变次序。从访问内存的角度看，这就使得 CPU 可能会有一些逻辑上已经完成、但是物理上尚未实现的内存操作。在 SMP 结构中，这种次序的改变也可能影响到 CPU 间的同步与互斥。所以，需要有一种手段，使得在某些操作之前把这种“欠下”的内存操作全都最终地、物理地完成，就好像把欠下的债都结清，然后再开始新的（通常是比较重要的）活动一样。这种手段称为“内存路障”(memory barrier)，对于 SMP 结构也是很重要的。Intel 在 Pentium 处理器中通过多种方式提供内存路障：

(1) 凡是对系统总线加锁的操作都起着内存路障的作用。所以，在 down( )中的 dec 指令就是一个内存路障，CPU 在执行这条指令之前会自动把已经写入高速缓存、但是尚未最终写回内存的内容先冲刷出去。

(2) 一些特殊的指令和操作起着内存路障的作用。这样的指令有 iret、 cpuid、 sfence，以及以控制寄存器或程序调试寄存器为目标的 mov 指令，还有对 GDTR、LDTR、IDTR 等寄存器的装入操作和对高速缓存的控制操作。这里特别值得一提的是指令 cpuid，我们有时候会在汇编代码中看到似乎与程序逻辑无关的 cpuid 指令，那其实就是起着路障的作用。

为了编写程序的方便，内核的代码中定义了几个用作路障的宏操作，具体有 mb( )、rmb( )以及 wmb( )，均定义于 include/asm-i386/system.h 内。对于 Intel 结构的 CPU，如 Pentium，rmb( )实际上就

是 mb( )。

```
257    /*
258     * Force strict CPU ordering.
259     * And yes, this is required on UP too when we're talking
260     * to devices.
261     *
262     * For now, "wmb( )" doesn't actually do anything, as all
263     * Intel CPU's follow what Intel calls a *Processor Order*,
264     * in which all writes are seen in the program order even
265     * outside the CPU.
266     *
267     * I expect future Intel CPU's to have a weaker ordering,
268     * but I'd also expect them to finally get their act together
269     * and add some real memory barriers if so.
270     *
271     * The Pentium III does add a real memory barrier with the
272     * sfence instruction, so we use that where appropriate.
273     */
274    #ifndef CONFIG_X86_XMM
275    #define mb( )   __asm__ __volatile__ \("lock; addl $0,0(%%esp)": : :"memory")
276    #else
277    #define mb( )        __asm__ __volatile__ ("sfence": : :"memory")
278    #endif
279    #define rmb( )  mb( )
280    #define wmb( )  __asm__ __volatile__ ("": : :"memory")
281
```

这里的指令 sfence 意为“存储器篱笆”(storage fence)，是专为存储器同步而设的。与 cpuid 相比，sfence 没有任何副作用，而 cpuid 则会改变 %eax 等寄存器的内容。但是，并不是所有的 Pentium CPU 都提供 sfence 指令，所以对不提供这条指令的 CPU 就通过一条带前缀 LOCK 的指令来实现，但是又要不产生任何副作用，所以把正在堆栈顶部的那个数据作为操作对象，在它上面加上 0。至于 wmb( )，代码的作者在注释中说目前 Intel 的 CPU 对写操作自动地保证“处理器序”，所以并不需要做什么事。这些函数还有个作用，就是使 gcc 在编译时不会试图跨过这些函数进行优化。

下面是一个在程序中调用函数 mb( )设置内存路障的实例，取自 kernel/softirq.c。

```
269    void init_bh(int nr, void (*routine)(void))
270    {
271        bh_base[nr] = routine;
272        mb( );
273    }
```

这里的目的显然是要使由当前 CPU 为某个软中断设置的 bh 函数立即为系统中其他 CPU 所见，其作用与冲刷 TLB 相似。

所以，对于现代的 CPU，不同处理器对临界资源操作的互斥性和次序的正确性主要是靠硬件解决

的，软件只起着辅助的作用。

顺便提一下，前面 bh_action( )的代码中引用了一个 inline 函数 smp_processor_id( )，目的是要知道当前进程在哪一个 CPU 上运行，与此相似的还有个 hard_smp_processor_id( )，均定义于 include/asm-i386/smp.h 内：

```
72    /*
73     * This function is needed by all SMP systems. It must _always_ be valid
74     * from the initial startup. We map APIC_BASE very early in page_setup( ),
75     * so this is correct in the x86 case.
76     */
77
78    #define smp_processor_id( ) (current->processor)
79
80    extern __inline int hard_smp_processor_id(void)
81    {
82        /* we don't want to mark this access volatile - bad code generation */
83        return GET_APIC_ID(*(unsigned long *)(APIC_BASE+APIC_ID));
84    }
```

每当一个 CPU 调度一个进程运行时，都把自己的逻辑序号设置在该进程的 task_struct 结构中的 processor 字段中。这个序号从哪儿来呢？很简单，来自调度之前原来在这个 CPU 上运行的进程。那么，第一个在此 CPU 上运行的进程(空转进程)又从哪里取得这个序号呢？对于主 CPU，这个序号是 0，而次 CPU 的序号则取决于主 CPU 启动它们运行的先后。至于 hard_smp_processor_id( )，则从本地 APIC 中读出其物理序号。

在单 CPU 系统中这两个函数都固定返回 0。

9.3　高速缓存与内存的一致性

本章第一节中讲到的第二个问题是各个 CPU 私有的、局部的高速缓存与公共的、全局的内存如何同步的问题。

对于高速缓存中的第一部分，实际上一般只有数据才有一致性的问题，因为对指令一般都是只读，而并不在运行的过程中动态地加以改变。Intel 在 Pentum CPU 中为已经装入高速缓存的数据提供了一种自动与内存保持一致的机制，称为“窥探”(snooping)。每个 CPU 内部都有一部分专门的硬件，一旦启用了高速缓存以后就时刻监视着系统总线上对内存的操作。由于对内存的操作一定要经过系统总线，所以没有一次实际访问内存的操作是能够逃过其监视的。如果发现有来自其他 CPU 的写操作，而本 CPU 的高速缓存中又缓冲存储着该次写操作的目标，就会自动把相应的缓冲线废弃，使得在需要用到这些数据时重新将其装入高速缓存，以此达到二者的一致。这样一来，SMP 结构中高速缓存与内存的数据一致性问题对于软件而言就是透明的了。显然，这种机制大大地简化了软件的设计与实现。读者可以回顾一下 SMP 结构中对临界资源加锁(见第 4 章)的过程。如果没有这种自动保持一致的机制，就必须把所有用于 spinlock 的变量全都放在一个不加缓冲的区间，否则各个 CPU 所测试和改变的可能只是自己高速缓存中的内容，而根本起不到 spinlock 的作用。

对于高速缓存中的 TLB 部分，问题便有所不同。这个问题可以通过 IPI、即“处理器间中断”来

解决。每当一个 CPU 改变了内存中某个页面映射目录或页面映射表的内容，从而可能引起其他 CPU 的 TLB 与此不一致时，就向系统中正在使用这个映射表的 CPU 发出一个中断请求，请它们废弃各自 TLB 中的内容。后面读者将会看到，在 i386 SMP 结构中采用的是“高级可编程中断控制器”APIC，在 APIC 中专门为此而设立了一种中断请求，其中断向量为 INVALIDATE_TLB_VECTOR。当一个 CPU 收到这种中断请求时，就会在相应的中断服务程序中将其 TLB 中的部分或全部内容作废(Invalidate)。TLB 中的内容作废以后，就会在需要用到这些内容时自动重新装入，这样就保证了内容的一致性。同时，内核中还提供了一个通用的函数 send_IPI_mask( )，当一个 CPU 需要向其他 CPU 发出某种中断请求时，就可以用相应的中断向量为参数调用这个函数来完成。对于 INVALIDATE_TLB_VECTOR 中断，内核还在 send_IPI_mask( )的基础上提供了一个函数 flush_tlb_others( )。当一个 CPU 要求其他 CPU 废弃各自 TLB 中的内容时，就可以调用 flush_tlb_others( )达到“冲刷其他(进程的)TLB”的目的。注意这里所谓“冲刷”实际上是废弃。

我们通过一个实例来看对 flush_tlb_others( )的调用。在第 2 章中，读者看到过内核线程 kswapd( )通过 try_to_swap_out( )试图换出某个进程的一个页面的过程。在函数 try_to_swap_out( )中，先对给定表项所映射的内存页面进行了种种检查。如果决定要断开这个页面的映射，就先把这个表项清成 0，然后再根据具体的情况决定是否需要将这个表项设置成指向盘上的页面映象。下面是这个函数中的一个片断(mm/vmscan.c)。

[kswapd( ) > do_try_to_free_pages( ) > refill_inactive ( ) > swap_out( ) > swap_out_mm( ) > swap_out_vma( ) > swap_out_pgd( ) > swap_out_pmd( ) > try_to_swap_out( )]

```
38    static int try_to_swap_out(struct mm_struct * mm, struct vm area struct* vma,
      unsigned long address, pte_t * page_table, int gfp_mask)
39    {
      . . . . . .
78        /* From this point on, the odds are that we're going to
79         * nuke this pte, so read and clear the pte.  This hook
80         * is needed on CPUs which update the accessed and dirty
81         * bits in hardware.
82         */
83        pte = ptep_get_and_clear(page table);
84        flush_tlb page(vma, address);
      . . . . . .
157   }
```

这里的 83 行通过 ptep_get_and_clear( )将指针 page_table 所指的表项清成 0，接着就调用 flush_tlb_page( )冲刷 CPU 的高速缓存。这里要说明：kswapd( )是个内核线程，正在某一个 CPU 上运行，而正在处理中的页面表却通常是属于另一个进程的，那个进程有可能正在另一个 CPU 上运行着。如果那样，则它的页面表很可能已被装入那个 CPU 的高速缓存中。所以，此时要向正在运行着目标进程的 CPU 发送一个 INVALIDATE_TLB_VECTOR 中断请求，让它废弃其高速缓存中的内容而重新装入已经改变了的页面映射表。这是由 flush_tlb_page( )完成的，其代码在 arch/i386/kernel/smp.c 中：

[kswapd( ) > do_try_to_free_pages( ) > refill_inactive ( ) > swap_out( ) > swap_out_mm( )> swap_out_vma( ) > swap_out_pgd( ) > swap_out_pmd( ) > try_to_swap_out( ) > flush_tlb_page( )]

```
372     void flush_tlb_page(struct vm_area_struct * vma, unsigned long va)
373     {
374         struct mm_struct *mm = vma->vm_mm;
375         unsigned long cpu_mask = mm->cpu_vm_mask & ~(1 << smp_processor_id( ));
376 
377         if (current->active_mm == mm) {
378             if(current->mm)
379                 __flush_tlb_one(va);
380              else
381                 leave_mm(smp_processor_id( ));
382         }
383 
384         if (cpu_mask)
385             flush_tlb_others(cpu_mask, mm, va);
386     }
```

读者应该还记得，进程的虚存空间是由一个 mm_struct 数据结构代表的。在 mm_struct 数据结构中有个字段 cpu_vm_mask，这是个位图，表示有哪一些 CPU 正在使用这个空间。系统中的每一个 CPU 都对应着这个位图中的一位，如果是单 CPU 系统则只有 bit0 有意义。当一个 CPU 在进程调度中从一个老进程切换到一个新进程的时候，就要修改这两个进程所使用的 mm_struct 数据结构中的位图，在新进程的 mm_struct 结构中把相应的标志位设置成 1，而老进程的 mm_struct 结构中则把相应的标志位设置成 0（详见 switch_mm( )的代码）。每个 CPU 都可以通过一个函数 smp_processor_id( )取得其在系统中的编号，这就决定了它在位图中的位置。以前，我们在第 4 章中阅读 switch_mm( )的代码时有意忽略了这个细节，现在读者可以回过去看一下。上面的 375 行把正在处理的 mm_struct 结构中的位图取到变量 cpu_mask 中，但是将正在执行这个函数的 CPU 本身除外。这样，如果 cpu_mask 为非 0 就说明系统中至少还有一个 CPU 正在运行使用这个 mm_struct 结构的进程，从而正在使用相应的页面表，所以要进一步调用 flush_tlb_others( )。至于代码中的 377～382 行，那只是在当前进程，就是正在当前 CPU 上运行的进程，恰好也与目标 mm_struct 结构有关的时候才执行，我们在这里对此并不关心。函数 flush_tlb_others( )的代码在 arch/i386/kernel/smp.c 中：

[kswapd( ) > do_try_to_free_pages( )> refill_inactive ( ) > swap_out( ) > swap_out_mm( ) > swap_out_vma( ) > swap_out_pgd( ) > swap_out_pmd( ) > try_to_swap_out( ) > flush_tlb_page( ) > flush_tlb_others( )]

```
304     static void flush_tlb_others (unsigned long cpumask, struct mm_struct *mm,
305                                 unsigned long va)
306     {
307         /*
308          * A couple of (to be removed) sanity checks:
309          *
310          * - we do not send IPIs to not-yet booted CPUs.
311          * - current CPU must not be in mask
312          * - mask must exist :)
313          */
314         if (!cpumask)
315             BUG( );
```

```
316         if ((cpumask & cpu_online_map) != cpumask)
317             BUG( );
318         if (cpumask & (1 << smp_processor_id( )))
319             BUG( );
320         if (!mm)
321             BUG( );
322 
323         /*
324          * i'm not happy about this global shared spinlock in the
325          * MM hot path, but we'll see how contended it is.
326          * Temporarily this turns IRQs off, so that lockups are
327          * detected by the NMI watchdog.
328          */
329         spin_lock(&tlbstate lock);
330 
331         flush_mm = mm;
332         flush_va = va;
333         atomic_set_mask(cpumask, &flush_cpumask);
334         /*
335          * We have to send the IPI only to
336          * CPUs affected.
337          */
338         send_IPI_mask(cpumask, INVALIDATE TLB VECTOR);
339 
340         while (flush_cpumask)
341             /* nothing. lockup detection does not belong here */;
342 
343         flush_mm = NULL;
344         flush_va = 0;
345         spin_unlock(&tlbstate_lock);
346     }
```

内核中有个全局量 cpu_online_map，这也是个位图，记录着系统中所有的 CPU（最多 32 个），参数 cpumask 的内容只能是 cpu_online_map 的一个子集，并且不能包括当前 CPU 本身。此外，flush_mm、flush_va 以及 flush_cpumask 都是全局量，用来作为 IPI 的辅助手段。由于是全局量，系统中所有的 CPU 都能看到这些变量。同时，对这些全局量的改变必须互斥。接着就是通过 send_IPI_mask( )向有关的 CPU 发送一个 INVALIDATE_TLB_VECTOR 中断请求，发送以后要等待位图 flush_cpumask 变成全 0，表示所有的目标 CPU 都已经响应了这次中断请求。我们把具体发送中断请求的过程留到下一节，这里先看看目标 CPU 在接收到这个中断请求以后干些什么。读者将会看到，与 INVALIDATE_TLB_VECTOR 相对应的中断响应程序是 smp_invalidate_interrupt( )，其代码也在 smp.c 中：

```
269     /*
270      * TLB flush IPI:
271      *
272      * 1) Flush the tlb entries if the cpu uses the mm that's being flushed.
273      * 2) Leave the mm if we are in the lazy tlb mode.
```

```
274     */
275
276    asmlinkage void smp_invalidate_interrupt (void)
277    {
278        unsigned long cpu = smp_processor_id( );
279
280        if (!test_bit(cpu, &flush_cpumask))
281            return;
282            /*
283             * This was a BUG( ) but until someone can quote me the
284             * line from the intel manual that guarantees an IPI to
285             * multiple CPUs is retried _only_ on the erroring CPUs
286             * its staying as a return
287             *
288             * BUG( );
289             */
290
291        if (flush_mm == cpu_tlbstate[cpu].active_mm) {
292            if (cpu_tlbstate[cpu].state == TLBSTATE_OK) {
293                if (flush_va == FLUSH_ALL)
294                    local_flush_tlb( );
295                else
296                    __flush_tlb_one(flush_va);
297            } else
298                leave_mm(cpu);
299        }
300        ack_APIC_irq( );
301        clear_bit(cpu, &flush_cpumask);
302    }
```

内核中有个全局的 tlb_state 数据结构数组 cpu_tlbstate[ ]，定义于 smp.c：

```
106    struct tlb_state cpu_tlbstate[NR_CPUS] = {[0 ... NR_CPUS-1] = { &init_mm, 0 }};
```

其类型定义在 include/asm-i386/pgalloc.h 中：

```
214    #define TLBSTATE_OK 1
215    #define TLBSTATE_LAZY   2
216
217    struct tlb_state
218    {
219        struct mm_struct *active_mm;
220        int state;
221    };
```

数组 cpu_tlbstate[ ]中所有元素的初值都是{ &init_mm, 0 }，106 行中的这种表示方法也是 gcc 对 C 语言的一种扩充。每个 CPU 在这个数组中都有一个 tlb_state 结构。在 switch_mm( )中，每当一个 CPU

要切换到一个进程的虚存空间时，就把这个结构中的指针 active_mm 设置成指向新的 mm_struct 结构，表示这个 CPU 正在使用这个虚存空间，并且把状态 state 设置成 TLBSTATE_OK，实际上是 1。在一般的情况下，一个 CPU 的 TLB 状态总是 TLBSTATE_OK，表示如果正在使用中的页面目录或页面表内容发生了变化就要刷新 TLB 的内容。

但是，是否只要页面目录或页面表内容发生了变化就一定要刷新 TLB 的内容呢？那倒不一定。如果有把握地知道可能发生的变化绝不会影响 CPU 的运行，就没有这个必要。我们不妨这样想：内核中代码所占的页面是不会改变的，其他的页面也没有换入/换出的问题，而内核线程又没有用户空间，所以与页面的换入/换出无关。事实上，内核中可能改变页面映射的只有几种情况，一种与 vmalloc( )有关，另一种与 HIGHMEM 的映射有关，还有就是与外设总线(如 PCI 总线)有关的映射。因此，只要一个内核线程与这些操作无关，那么这个内核线程就可以“任凭风浪起，稳坐钓鱼船”。所以，在一些特殊的情况下，CPU 虽然还在使用属于某个虚拟空间的页面目录或页面表，但是即使这些页面目录或页面表发生了变化也没有必要刷新 TLB 的内容。例如，在执行系统调用 exit( )的过程中，即使当前进程的页面目录或页面表发生了变化，也已经没有必要更新 TLB 的内容了。还有一种情况是，当 CPU 切换到一个不具有用户空间的内核线程时，要借用在它之前运行的那个进程的 active_mm（详见“进程的调度与切换”），所以此时进程切换了，但是页面目录和页面表却没有切换。然而，在运行这个内核线程的期间，即使用户空间的页面目录或页面表发生了变化也没有必要更新 TLB 的内容，因为内核线程本来就没有用户空间。在这样的情况下，就通过一个 inline 函数 enter_lazy_tlb( )，将当前 CPU 的 TLB 状态设置成 TLBSTATE_LAZY，表示懒得更新。这个函数的代码在 include/asm-i386/mmu_context.h 中：

```
17	static inline void enter_lazy_tlb(struct mm struct *mm,
	                                  struct task struct *tsk, unsigned cpu)
18	{
19		if(cpu_tlbstate[cpu].state == TLBSTATE_OK)
20			cpu_tlbstate[cpu].state = TLBSTATE_LAZY;
21	}
```

调用这个函数的地方主要有两处，一处是在__exit_mm( )中，还有一处是在 schedule( )中，其他是在初始化阶段，那时候运行的也是内核线程。

对 TLB 的冲刷（其实是废弃）可以是针对整个 TLB 的，也可以是针对一个具体页面的。对整个 TLB 的冲刷由 local_flush_tlb( )进行，这是个宏操作，定义于 include/asm-i386/pgalloc.h 中：

```
199	#define local_flush_tlb( ) \
200		__flush_tlb( )
```

这里的__flush_tlb( )又是个宏操作，其定义也在同一文件中：

```
37	#define __flush_tlb( )                            \
38		do {                                          \
39			unsigned int tmpreg;                      \
40	                                                  \
41			__asm__ volatile__(                       \
42				"movl %%cr3, %0;  # flush TLB \n"     \
43				"movl %0, %%cr3;              \n"     \
```

```
44                 : "=r" (tmpreg)                  \
45                 :: "memory");                    \
46         } while (0)
```

要废弃整个 TLB 的内容很简单，只要把指向映射目录的指针，即控制寄存器%cr3 重新装入一次就可以了。所以这里把%cr3 中的内容先读到临时变量 tmpreg 中，然后再把它写回%cr3 就成了。虽然这个操作的前后%cr3 的内容并无改变，但是对%cr3 的写操作本身就起到了废弃整个 TLB 的内容，从新装入页面目录的作用。

再看废弃某个具体页面的操作，这是由宏操作__flush_tlb_one( )完成的，定义于同一文件中：

```
87     #define __flush_tlb_one(addr) \
88     __asm__ __volatile__("invlpg %0": :"m" (*(char *) addr))
```

指令 invlpg 将 TLB 中属于给定页面的内容废弃。TLB 是按 32 字节“缓冲线”来缓冲的，所以给定页面的内容未必全都在 TLB 中，在 TLB 中的内容也未必连续。这条指令的作用就是把 TLB 中凡是属于给定页面的内容全都废弃。

如果 CPU 当前的 TLB 状态是 TLBSTATE_LAZY，那就干脆“闭目塞听”，通过 leave_mm( )退出当前 mm_struct 结构中的位图 cpu_vm_mask，以后再有类似的改变就不会向这个 CPU 发出中断请求了。这个函数的代码在 arch\i386\kernel\smp.c 中：

[smp_invalidate_interrupt( ) > leave_mm( )]

```
219    /*
220     * We cannot call mmdrop( ) because we are in interrupt context,
221     * instead update mm->cpu_vm_mask.
222     */
223    static void inline leave_mm (unsigned long cpu)
224    {
225        if (cpu_tlbstate[cpu].state == TLBSTATE_OK)
226            BUG( );
227        clear_bit(cpu, &cpu_tlbstate[cpu].active_mm->cpu_vm_mask);
228    }
```

所以，如果当前 CPU 的 tlb_state 数据结构中的指针指向某个 mm_struct 结构，而这个结构内的位图 cpu_vm_mask 又不包括当前 CPU，那就说明这个 CPU 的 TLB 实际上已经不一致了，只不过因为当前进程是内核线程，这种不一致不会引起什么后果而已。到下一次进程调度的时候，如果要切换到一个使用着不同页面目录的进程，就会在切换时装入新的页面目录，并且把当前 CPU 的 TLB 状态又改为 TLBSTATE_OK，这样，事情就偷懒过去了。但是，如果新的进程使用的恰好就是这个内核线程所借用的 mm_struct 结构呢？在单 CPU 结构的系统中此时不用做任何事，因为不需要切换虚存空间，TLB 中的内容又肯定是一致的。而在 SMP 结构的系统中，这时候就要检查一下了，如果发生了上述的情况就得补上一次 TLB 刷新，因为页面目录或页面表中已经发生的变化对即将运行的进程是有影响的。下面是 inline 函数 switch_mm( )中的一个片段，在 include/asm-i386/mmu_context.h 中：

```
28     static inline void switch_mm(struct mm_struct *prev,
```

```
            struct mm_struct *next, struct task_struct *tsk, unsigned cpu)
29    {
30        if (prev != next) {
      . . . . . .
45        }
46    #ifdef CONFIG_SMP
47        else {
48            cpu_tlbstate[cpu].state = TLBSTATE_OK;
49            if(cpu_tlbstate[cpu].active_mm != next)
50                BUG( );
51            if(!test_and_set_bit(cpu, &next->cpu_vm_mask)) {
52                /* We were in lazy tlb mode and leave_mm disabled
53                 * tlb flush IPI delivery. We must flush our tlb.
54                 */
55                local_flush_tlb( );
56            }
57        }
58    #endif
59    }
```

读者不妨回到第 4 章中，带着问题再读一下有关的代码。

回到 smp_invalidate_interrupt( )的代码中。最后，通过 ack_APIC_irq( )向 APIC 发出一个确认，然后把当前 CPU 在位图 flush_cpumask 中的对应位清 0，使发出中断请求的 CPU 能知道当前 CPU 已经废弃了 TLB 中的内容。另一方面，发出中断请求的 CPU 此时正在 flush_tlb_others( )中通过一个 while 循环等待位图 flush_cpumask 变成全 0，当所有的目标 CPU 都完成了中断服务时，flush_tlb_others ( )的执行也就完成了。

我们在前面 flush_tlb_page( )的代码中跳过了对本地 TLB 的“冲刷”，但是从代码中可以看出，具体的处理同样也是由__flush_tlb( )或 leave_mm( )完成的。

在 try_to_swap_out( )中之所以调用 flush_tlb_page( )，是因为要改变了一个页面映射表的内容，而一个页面映射表正好占一个页面，所以只要废弃 TLB 中属于这个页面的内容就可以了。可是，如果改变的是页面映射目录的内容呢？虽然页面映射目录也占一个页面，但是映射目录的改变意味着整个映射的改变，因为目录中的每一项都指向一个页面映射表。所以，仅仅废弃映射目录所在的页面是不够的，此时需要废弃的是 TLB 中的全部内容。为了这个目的，内核中还有个函数 flush_tlb_mm( )，用来废弃整个 TLB 的内容，其代码在 arch/i386/kernel/smp.c 中：

```
358   void flush_tlb_mm (struct mm_struct * mm)
359   {
360       unsigned long cpu_mask = mm->cpu_vm_mask & ~(1 << smp_processor_id( ));
361
362       if (current->active_mm == mm) {
363           if (current->mm)
364               local_flush_tlb( );
365           else
366               leave_mm(smp_processor_id( ));
367       }
```

```
368         if (cpu_mask)
369             flush_tlb_others(cpu_mask, mm, FLUSH_ALL);
370     }
```

代码中的 local_flush_tlb( )实际上就是__flush_tlb( )，定义于 include/asm-i386/pgalloc.h：

```
199     #define local_flush_tlb( ) \
200         __flush_tlb( )
```

将 flush_tlb_mm( )与 flush_tlb_page( )作一比较，就可以发现二者基本上是相同的，只是调用 flush_tlb_others( )时的一个参数为 FLUSH_ALL，而不是具体的地址。前面我们已经看到，当参数为 FLUSH_ALL 时各个 CPU 都通过__flush_tlb( )重新装入控制寄存器%cr3，这就起到了废弃 TLB 中全部内容，然后随着运行的需要重新装入的目的。

9.4　SMP 结构中的中断机制

传统的 i386 处理器都采用 8259A 中断控制器，读者在第 3 章看到过对这种器件的初始化。一般而言，8259A 的作用是提供多个外部中断源与单一 CPU 之间的连接。如果在 SMP 结构中还是采用 8259A 中断控制器，那就只能静态地把所有的外部中断源划分成若干组，分别把每一组都连接到一个 8259A，而 8259A 则与 CPU 有一对一的连接。然而这样，就达不到动态地分配中断请求的目的，也使硬件的设计变得很不简洁。因此，Intel 为 Pentiun 处理器设计了一种更为通用的中断控制器，称为“高级可编程中断控制器”(Advanced Programable Interrupt Controllor)，缩写成 APIC。另一方面，考虑到“处理器间中断请求”的需要，每个 CPU 实际上都需要有个本地的 APIC，因为一个 CPU 常常要有目标地向系统中的其他 CPU 发出中断请求，所以，从 Pentium 开始，Intel 就在 CPU 芯片内部集成了一个本地 APIC。但是，在 SMP 结构中还需要一个外部的、全局的 APIC，从而形成如图 9.2 所示的结构。

一般而言，集成在 CPU 芯片内部的本地 APIC 与外部的 I/O APIC 是配合使用的，实际上可以认为是将一个器件分成了两部分，既可以用于 SMP 结构，也可以用于单 CPU 结构。另一方面，在本地 APIC 中有个可以用于时钟中断源的定时器，所以即使没有 I/O APIC 的配合，也可以选择使用本地 APIC。而且，即使采用了 APIC，仍然可以把各个 CPU 单独连接到 8259A 中断控制器，图 9.2 中的“本地中断请求”就是指来自各自外部中断控制器的中断请求，以及由 CPU 内部产生的中断请求（如陷阱）。在内核的代码中，因采用本地 APIC 而需要的代码都放在条件编译选择项 CONFIG_X86_LOCAL_APIC 下面。

读者在第 3 章中曾经看到，内核中的中断响应程序是通过一些宏操作利用 gcc 预处理的字符串替换和拼接功能自动生成的。同样，几个为 SMP 结构专用的中断响应程序也是由这样的宏操作生成的，这个宏操作定义于 include/asm-i386/hw-irq.h 中：

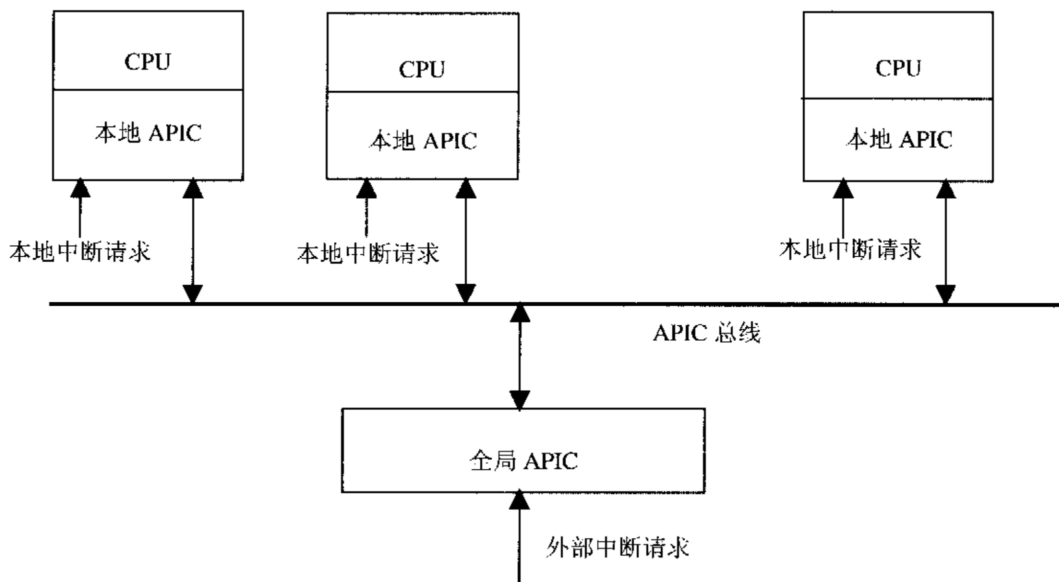


图 9.2 SMP 结构中的中断控制机构

```
123     #define BUILD_SMP_INTERRUPT(x,v) XBUILD_SMP_INTERRUPT(x,v)
124     #define XBUILD_SMP_INTERRUPT(x,v)\
125     asmlinkage void x(void); \
126     asmlinkage void call_##x(void); \
127     __asm__( \
128     "\n"__ALIGN_STR"\n" \
129     SYMBOL_NAME_STR(x) ":\n\t" \
130         "pushl $"#v"\n\t" \
131         SAVE_ALL \
132         SYMBOL_NAME_STR(call_##x)":\n\t" \
133         "call "SYMBOL_NAME_STR(smp_##x)"\n\t" \
134         "jmp ret_from_intr\n");
```

文件 arch/i386/kernel/i8259.c 中引用了这个宏操作：

```
74      /*
75       * The following vectors are part of the Linux architecture, there
76       * is no hardware IRQ pin equivalent for them, they are triggered
77       * through the ICC by us (IPIs)
78       */
79      #ifdef CONFIG_SMP
80      BUILD_SMP_INTERRUPT(reschedule_interrupt,RESCHEDULE_VECTOR)
81      BUILD_SMP_INTERRUPT(invalidate_interrupt,INVALIDATE_TLB_VECTOR)
82      BUILD_SMP_INTERRUPT(call_function_interrupt,CALL_FUNCTION_VECTOR)
83      #endif
```

以这里的 80 行为例，经过 gcc 的预处理以后就会变成这样：

```
asmlinkage void reschedule_interrupt(void); \
asmlinkage void call_reschedule_interrupt(void); \
__asm__(
reschedule_interrupt:
        pushl $RESCHEDULE_VECTOR
        SAVE_ALL
call_smp_reschedule_interrupt:
        call smp_reschedule_interrupt
        jmp ret_from_intr
)
```

因此，当发生 RESCHEDULE_VECTOR 中断时，响应程序的入口是 reschedule_interrupt( )，而实际处理中断响应的函数则为 smp_reschedule_interrupt( )。

同样的道理，与 INVALIDATE_TLB_VECTOR 相对应的入口是 invalidate_interrupt( )，而实际处理中断响应的则是 smp_invalidate_interrupt( )；与 CALL_FUNCTION_VECTOR 相对应的入口是 call_function_interrupt( )，而实际处理中断响应的是 smp_call_function_interrupt( )。

我们在前一节中已经看过 invalidate_interrupt( )的代码，这里再来看看另一个中断服务程序 smp_reschedule_interrupt( )的代码。顾名思义，这个函数使 CPU 应系统中另一 CPU 之请求而进行一次进程调度，其代码在 arch/i386/kernel/smp.c 中：

```
513     /*
514      * Reschedule call back. Nothing to do,
515      * all the work is done automatically when
516      * we return from the interrupt.
517      */
518     asmlinkage void smp_reschedule_interrupt(void)
519     {
520         ack_APIC_irq();
521     }
```

从表面上看，这个函数所做的似乎只是通过 ack_APIC_irq( )向 CPU 中的 APIC 发出对中断请求的确认，其代码定义于 include/asm-i386/apic.h：

```
54      extern inline void ack_APIC_irq(void)
55      {
56          /*
57           * ack_APIC_irq() actually gets compiled as a single instruction:
58           * - a single rmw on Pentium/82489DX
59           * - a single write on P6+ cores (CONFIG_X86_GOOD_APIC)
60           * ... yummie.
61           */
62
63          /* Docs say use 0 for future compatibility */
64          apic_write_around(APIC_EOI, 0);
65      }
```

就是说，往本地 APIC 的一个寄存器中写一个 0，表示已经收到了中断请求，如此而已。可是，实际上对这种中断请求的服务隐藏在内核对中断处理的公共部分。读者在第 3 章中看到，不管是什么中断请求，内核在针对特定中断请求的服务完成以后都要检查（本 CPU）是否应该进行一次进程调度，而这正是 smp_reschedule_interrupt( )所要达到的目的。当一个 CPU 需要另一个 CPU 进行一次进程调度时，就可以通过调用一个函数 smp_send_reschedule( )向目标 CPU 发出一个 RESCHEDULE_VECTOR 中断请求，这个函数在 arch/i386/kernel/smp.c 中：

```
415    void smp_send_reschedule(int cpu)
416    {
417        send_IPI_mask(1 << cpu, RESCHEDULE_VECTOR);
418    }
```

中断请求的直接目的并不在于让对方进行一次进程调度，而只是使对方产生一次中断。至于目标 CPU 在执行完中断服务程序以后是否进行进程调度，那得要看事先或者处理中断的过程中是否将当前进程的 need_resched 标志设置成了 1。

相应地，在系统初始化时要在函数 init_IRQ( )中为这几个中断设置好相应的中断门。我们在第 3 章中看过函数 init_IRQ( )的代码，当时忽略了与 SMP 结构有关的部分，现在应该补上了。这个函数在 arch/i386/kernel/i8259.c 中：

```
438    void __init init_IRQ(void)
439    {
440        int i;
441
442    #ifndef CONFIG_X86_VISWS_APIC
443        init_ISA_irqs( );
444    #else
445        init_VISWS_APIC_irqs( );
446    #endif
447        /*
448         * Cover the whole vector space, no vector can escape
449         * us. (some of these will be overridden and become
450         * 'special' SMP interrupts)
451         */
452        for (i = 0; i < NR_IRQS; i++) {
453            int vector = FIRST_EXTERNAL_VECTOR + i;
454            if (vector != SYSCALL_VECTOR)
455                set_intr_gate(vector, interrupt[i]);
456        }
457
458    #ifdef CONFIG_SMP
459        /*
460         * IRQ0 must be given a fixed assignment and initialized,
461         * because it's used before the IO-APIC is set up.
462         */
```

```
463         set_intr_gate(FIRST_DEVICE_VECTOR, interrupt[0]);
464 
465         /*
466          * The reschedule interrupt is a CPU-to-CPU reschedule-helper
467          * IPI, driven by wakeup.
468          */
469         set_intr_gate(RESCHEDULE_VECTOR, reschedule_interrupt);
470 
471         /* IPI for invalidation */
472         set_intr_gate(INVALIDATE_TLB_VECTOR, invalidate_interrupt);
473 
474         /* IPI for generic function call */
475         set_intr_gate(CALL_FUNCTION_VECTOR, call_function_interrupt);
476     #endif
477 
478     #ifdef CONFIG_X86_LOCAL_APIC
479         /* self generated IPI for local APIC timer */
480         set_intr_gate(LOCAL_TIMER_VECTOR, apic_timer_interrupt);
481 
482         /* IPI vectors for APIC spurious and error interrupts */
483         set_intr_gate(SPURIOUS_APIC_VECTOR, spurious_interrupt);
484         set_intr_gate(ERROR_APIC_VECTOR, error_interrupt);
485     #endif
486 
487         /*
488          * Set the clock to HZ Hz, we already have a valid
489          * vector now:
490          */
491         outb_p(0x34,0x43);        /* binary, mode 2, LSB/MSB, ch 0 */
492         outb_p(LATCH & 0xff , 0x40);     /* LSB */
493         outb(LATCH >> 8 , 0x40);     /* MSB */
494 
495     #ifndef CONFIG_VISWS
496         setup_irq(2, &irq2);
497     #endif
498 
499         /*
500          * External FPU? Set up irq13 if so, for
501          * original braindamaged IBM FERR coupling.
502          */
503         if (boot_cpu_data.hard_math && !cpu_has_fpu)
504             setup_irq(13, &irq13);
505     }
```

代码中 452 行的 for 循环原已设置了从 FIRST_EXTERNAL_VECTOR，即 0x20 开始的 224 个中断门向量，而如果是 SMP 系统就修改了其中的 4 个，它们分别对应着下列 4 个中断向量（见 include/asm-i386/hw_irq.h）：

```
41      #define INVALIDATE_TLB_VECTOR    0xfd
42      #define RESCHEDULE_VECTOR        0xfc
43      #define CALL_FUNCTION_VECTOR     0xfb

52      /*
53       * First APIC vector available to drivers: (vectors 0x30-0xee)
54       * we start at 0x31 to spread out vectors evenly between priority
55       * levels. (0x80 is the syscall vector)
56       */
57      #define FIRST_DEVICE_VECTOR      0x31
58      #define FIRST_SYSTEM_VECTOR      0xef
```

为中断向量 FIRST_DEVICE_VECTOR 设置的中断响应程序是 interrupt[0]，回顾一下第 3 章中的有关内容便可以知道，这个程序是 IRQ0x01_interrupt( ).进入这个函数以后就会把数值－256，即 0xffffff00 压入堆栈，然后转入 common_interrupt，最后进入 do_IRQ( )，余可类推。其中从 0x31 到 0xef 是用于外部 APIC 即 I/O APIC 的中断向量。这个区间的中断向量基本上没有什么变化，还与采用 8259A 时大致相同。然而，在 SMP 结构中这些中断请求将要由外部 APIC（而不是 8259A）送达各个 CPU，所以还要通过一个函数 setup_IO_APIC( )对外部 APIC 芯片进行初始化，并关闭 8259A。与此有关的代码都在 arch/i386/kernel/io_apic.c 中。因代码过于专门，我们在这里就从略了。有兴趣或需要的读者可结合 Intel 的外部 APIC 芯片（82489DX）技术资料阅读这些代码。

此外，479～484 行为 CPU 的内部 APIC 设置中断向量，主要是时钟中断。在单 CPU 的系统中，是否采用内部 APIC 是个编译选择项。

不管是外部还是内部，APIC 都支持从 0x20 到 0xff 共 240 个不同的中断向量（0～0x1f 用于 CPU 本身的陷阱）。这些中断向量分成 15 个优先级，可以按中断向量号除于 16 算得，优先级 15 为最高。每个 CPU 都通过设置其内部 APIC 表明准备响应哪一些中断请求。不过，每个 CPU 都应将准备加以响应的中断向量尽量均匀地分布于不同的优先级中，Intel 的技术资料规定每个 CPU 在每个优先级中使用的中断向量不超过两个，否则就有可能丢失中断请求。

外部 APIC 担负着把来自外部设备的中断请求提交和分配给系统中各个 CPU 的任务。对于每一个中断向量，可以将外部 APIC 设置成静态或动态两种不同模式之一。如果某一个中断向量是静态分配的，则外部 APIC 把这种中断请求提交给预设的一个或多个 CPU；否则就提交给优先级最低的 CPU。而各个 CPU 当前的优先级也是可以通过内部 APIC 设置的。与此有关的代码都在 arch/i386/kernel/apic.c 里，这些代码也过于专门，我们就不深入进去了。外部 APIC 的初始化是由主 CPU 通过 setup_IO_APIC( ) 进行的，其代码在 arch/i386/kernel/io_apic.c 中：

```
1521    void __init setup_IO_APIC(void)
1522    {
1523        enable_IO_APIC( );
1524
1525        io_apic_irqs = ~PIC_IRQS;
1526        printk("ENABLING IO-APIC IRQs\n");
1527
1528        /*
1529         * Set up the IO-APIC IRQ routing table by parsing the MP-BIOS
```

```
1530         * mptable:
1531         */
1532        setup_ioapic_ids_from_mpc( );
1533        sync_Arb_IDs( );
1534        setup_IO_APIC_irqs( );
1535        init_IO_APIC_traps( );
1536        check_timer( );
1537        print_IO_APIC( );
1538    }
```

考虑到本书的篇幅，我们无法在这里深入到这些代码中了。事实上，光是与 APIC 有关的内容和代码就已经够写一本书了。

除外部 APIC 可以把来自外部设备的中断请求提交系统中的各个 CPU 外，每个 CPU 也都可以通过其内部 APIC 向其他 CPU 发出中断请求。当一个 CPU 要引起其他 CPU 的 INVALIDATE_TLB_VECTOR 或 RESCHEDULE_VECTOR 中断时，可以通过调用 send_IPI_mask( )来达到目的。这个函数的代码在 arch/i386/kernel/smp.c 中：

```
172    static inline void send_IPI_mask(int mask, int vector)
173    {
174        unsigned long cfg;
175        unsigned long flags;
176
177        __save_flags(flags);
178        __cli( );
179
180        /*
181         * Wait for idle.
182         */
183        apic_wait_icr_idle( );
184
185        /*
186         * prepare target chip field
187         */
188        cfg = __prepare_ICR2(mask);
189        apic_write_around(APIC_ICR2, cfg);
190
191        /*
192         * program the ICR
193         */
194        cfg = __prepare_ICR(0, vector);
195
196        /*
197         * Send the IPI. The write to APIC_ICR fires this off.
198         */
199        apic_write_around(APIC_ICR, cfg);
```

```
200         __restore_flags(flags);
201     }
```

CPU 的内部 APIC 有一些控制寄存器，APIC_ICR 和 APIC_ICR2 是其中的两个。要向系统中的某一个 CPU 发出中断请求时，首先要通过 apic_wait_icr_idle( )，确认或等待 APIC_ICR 处于空闲状态，然后通过 __prepare_ICR( ) 和 __prepare_ICR2( )，准备好要写入这两个寄存器的数值(arch/i386/kernel/smp.c)：

```
114     static inline int __prepare_ICR (unsigned int shortcut, int vector)
115     {
116         return APIC_DM_FIXED | shortcut | vector | APIC_DEST_LOGICAL;
117     }
118
119     static inline int __prepare_ICR2 (unsigned int mask)
120     {
121         return SET_APIC_DEST_FIELD(mask);
122     }
```

寄存器 APIC_ICR2 主要用来说明发送中断请求的目标，这种目标可以是具体的 CPU，也可以是除发送者自身以外的所有 CPU，还可以是包括发送者自身在内的所有 CPU，甚至仅是发送者自身。最后，把含有发送目标的数值写入寄存器 APIC_ICR2，把含有中断向量（如 RESCHEDULE_VECTOR）的数值写入寄存器 APIC_ICR，就完成了中断请求的发送操作。

还有一个处理器间中断向量是 CALL_FUNCTION_VECTOR，用来请求目标 CPU 执行一个指定的函数。发送者先设置好一个全局的 call_data_struct 数据结构，然后向目标 CPU 发出请求。这个数据结构的定义在 arch/i386/kernel/smp.c 中：

```
420     /*
421      * Structure and data for smp call_function( ). This is designed to minimise
422      * static memory requirements. It also looks cleaner.
423      */
424     static spinlock_t call_lock = SPIN_LOCK_UNLOCKED;
425
426     struct call_data_struct {
427         void (*func) (void *info);
428         void *info;
429         atomic_t started;
430         atomic_t finished;
431         int wait;
432     };
433
434     static struct call_data_struct * call_data;
```

数据结构中的函数指针 func 就是要求对方执行的函数，另一个指针 info 则为参数。读者在前面已经看到，对应于 CALL_FUNCTION_VECTOR 的中断服务程序是 smp_call_function_interrupt( )，其代码在 arch/i386/kernel/smp.c 中：

```
523    asmlinkage void smp_call_function_interrupt(void)
524    {
525        void (*func) (void *info) = call_data->func;
526        void *info = call_data->info;
527        int wait = call data->wait;
528
529        ack_APIC_irq( );
530        /*
531         * Notify initiating CPU that I've grabbed the data and am
532         * about to execute the function
533         */
534        atomic_inc(&call_data->started);
535        /*
536         * At this point the info structure may be out of scope unless wait==1
537         */
538        (*func)(info);
539        if (wait)
540            atomic_inc(&call_data->finished);
541    }
```

当然，一般的函数是没有必要请其他 CPU 来执行的，系统中所有的 CPU 都可共享着同样的代码和数据。之所以要通过处理器间中断请其他 CPU 执行，是因为某个函数非得由目标 CPU 才能完成。例如，Pentium 处理器有一条指令 cpuid，通过这条指令可以查询本 CPU 是什么型号、什么版本，是否支持一些特殊的功能，当前的功能设置等等的信息。但是这条指令只能由具体的 CPU 本身执行，而不能由别的 CPU 代替。这样，如果要知道系统中某一个 CPU 的有关情况，就只能通过这种处理器间中断来实现。具体的函数为 cpuid_smp_cpuid( )，其代码在 arch/i386/kernel/cpuid.c 中，读者可以自己阅读。

有时候需要把中断请求发送给除当前 CPU 自身以外的所有 CPU，此时可以通过 send_IPI_allbutself( )，发出广播式的中断请求。其代码在 arch/i386/kernel/smp.c 中：

```
151    static inline void send_IPI_allbutself(int vector)
152    {
153        /*
154         * if there are no other CPUs in the system then
155         * we get an APIC send error if we try to broadcast.
156         * thus we have to avoid sending IPIs in this case.
157         */
158        if (smp num_cpus > 1)
159            __send_IPI_shortcut(APIC_DEST_ALLBUT, vector);
160    }
```

具体的发送操作由__send_IPI_shortcut( )完成，其代码也在同一文件中：

[send_IPI_allbutself( ) > __send_IPI_shortcut( )]

```
124    static inline void __send_IPI_shortcut(unsigned int shortcut, int vector)
125    {
```

```
126         /*
127          * Subtle. In the case of the 'never do double writes' workaround
128          * we have to lock out interrupts to be safe.  As we don't care
129          * of the value read we use an atomic rmw access to avoid costly
130          * cli/sti.  Otherwise we use an even cheaper single atomic write
131          * to the APIC.
132          */
133         unsigned int cfg;
134 
135         /*
136          * Wait for idle.
137          */
138         apic_wait_icr_idle( );
139 
140         /*
141          * No need to touch the target chip field
142          */
143         cfg = __prepare_ICR(shortcut, vector);
144 
145         /*
146          * Send the IPI. The write to APIC_ICR fires this off.
147          */
148         apic_write_around(APIC_ICR, cfg);
149     }
```

可见，广播式的中断请求发送操作要略为简单一些。

在所有的中断请求中，时钟中断扮演着特殊的重要角色。以前讲过，人们把时钟中断比喻作“心跳”。在 SMP 结构的系统中，内部 APIC 本身就包含了一个 32 位可编程定时器，所以各个 CPU 可以选择使用本地的时钟中断源，也可以选择使用外部的全局时钟中断源。所谓可编程定时器实质上是个计数器，设置好计数器的初值以后，每来一个时钟脉冲计数器就减 1，减到 0 时就发出一个中断请求。在系统初始化阶段，主 CPU 在启动次 CPU 运行以后便通过 setup_APIC_clocks( )设置 APIC 中的时钟中断源，这个函数的代码在 arch/i386/kernel/apic.c 中：

[start_kernel( ) > smp_init( ) > smp_boot_cpus( ) > setup_APIC_clocks( )]

```
597     void __init setup_APIC_clocks (void)
598     {
599         __cli( );
600 
601         calibration_result = calibrate_APIC_clock( );
602         /*
603          * Now set up the timer for real.
604          */
605         setup_APIC_timer((void *)calibration_result);
606 
607         __sti( );
```

```
608
609         /* and update all other cpus */
610         smp_call_function(setup_APIC_timer, (void *)calibration_result, 1, 1);
611     }
```

在此之前，系统先设置了一个外部时钟中断源供所有 CPU 共享，并且以此为基准测算各个 CPU 的运算速度。到初始化基本完成时再通过这个函数为各个 CPU 设置其内部 APIC 中的定时器，以作为本地的时钟中断源。这里，先通过 calibrate_APIC_clock( )，测算出时钟中断相对于 APIC 总线上时钟脉冲的周期。所有 CPU 的 APIC 都通过 APIC 总线连在一起，它们都有相同的时钟脉冲周期，根据这个时钟脉冲周期就可以计算出时钟中断的周期。具体的设置是由 setup_APIC_timer( )完成的。当然，各个 CPU 只能设置其自己的 APIC，而不能直接设置其他 CPU 的 APIC，所以通过 smp_call_function( ) 向所有的次 CPU 都发出一个处理器间中断请求，让各个次 CPU 也来执行这个函数。其函数代码也在 arch/i386/kernel/apic.c 中：

```
469     void setup_APIC_timer(void * data)
470     {
471         unsigned int clocks = (unsigned int) data, slice, t0, t1;
472         unsigned long flags;
473         int delta;
474
475         __save_flags(flags);
476         __sti( );
477         /*
478          * ok, Intel has some smart code in their APIC that knows
479          * if a CPU was in 'hlt' lowpower mode, and this increases
480          * its APIC arbitration priority. To avoid the external timer
481          * IRQ APIC event being in synchron with the APIC clock we
482          * introduce an interrupt skew to spread out timer events.
483          *
484          * The number of slices within a 'big' timeslice is smp_num_cpus+1
485          */
486
487         slice = clocks / (smp_num_cpus+1);
488         printk("cpu: %d, clocks: %d, slice: %d\n",
489             smp_processor_id( ), clocks, slice);
490
491         /*
492          * Wait for IRQ0's slice:
493          */
494         wait_8254_wraparound( );
495
496         __setup_APIC_LVTT(clocks);
497
498         t0 = apic_read(APIC_TMICT)*APIC_DIVISOR;
499         /* Wait till TMCCT gets reloaded from TMICT... */
500         do {
```

```
501             t1 = apic_read(APIC_TMCCT)*APIC_DIVISOR;
502             delta = (int)(t0 - t1 - slice*(smp_processor_id( )+1));
503         } while (delta >= 0);
504         /* Now wait for our slice for real. */
505         do {
506             t1 = apic_read(APIC_TMCCT)*APIC_DIVISOR;
507             delta = (int)(t0 - t1 - slice*(smp_processor_id( )+1));
508         } while (delta < 0);
509
510           setup_APIC_LVTT(clocks);
511
512         printk("CPU%d<T0:%d,T1:%d,D:%d,S:%d,C:%d>\n",
513                 smp_processor_id( ), t0, t1, delta, slice, clocks);
514
515         __restore_flags(flags);
516     }
```

为了不让所有的 CPU 都在同一时刻发生时钟中断，这里通过两个 do-while 循环根据逻辑 CPU 号引入不同数量的延迟，使各个 CPU 的时钟中断在相位上互相错开，使这些中断均匀地分布在时钟中断的周期中。

我们把各个 CPU 对本地时钟中断的服务程序列出于下(arch/i386/kernel/apic.c)，供读者结合第 3 章中与时钟中断有关的内容自行阅读。

```
709     void smp_apic_timer_interrupt(struct pt_regs * regs)
710     {
711         int cpu = smp_processor_id( );
712
713         /*
714          * the NMI deadlock-detector uses this.
715          */
716         apic_timer_irqs[cpu]++;
717
718         /*
719          * NOTE! We'd better ACK the irq immediately,
720          * because timer handling can be slow.
721          */
722         ack_APIC_irq( );
723         /*
724          * update_process_times( ) expects us to have done irq_enter( ).
725          * Besides, if we don't timer interrupts ignore the global
726          * interrupt lock, which is the WrongThing (tm) to do.
727          */
728         irq_enter(cpu, 0);
729         smp_local_timer_interrupt(regs);
730         irq_exit(cpu, 0);
731     }
```

函数 smp_local_timer_interrupt( )的代码也在 arch/i386/kernel/apic.c 中:

```
643    /*
644     * Local timer interrupt handler. It does both profiling and
645     * process statistics/rescheduling.
646     *
647     * We do profiling in every local tick, statistics/rescheduling
648     * happen only every 'profiling multiplier' ticks. The default
649     * multiplier is 1 and it can be changed by writing the new multiplier
650     * value into /proc/profile.
651     */
652
653    inline void smp_local_timer_interrupt(struct pt_regs * regs)
654    {
655        int user = user_mode(regs);
656        int cpu = smp_processor_id( );
657
658        /*
659         * The profiling function is SMP safe. (nothing can mess
660         * around with "current", and the profiling counters are
661         * updated with atomic operations). This is especially
662         * useful with a profiling multiplier != 1
663         */
664        if (!user)
665            x86_do_profile(regs->eip);
666
667        if (--prof_counter[cpu] <= 0) {
668            /*
669             * The multiplier may have changed since the last time we got
670             * to this point as a result of the user writing to
671             * /proc/profile. In this case we need to adjust the APIC
672             * timer accordingly.
673             *
674             * Interrupts are already masked off at this point.
675             */
676            prof_counter[cpu] = prof_multiplier[cpu];
677            if (prof_counter[cpu] != prof_old_multiplier[cpu]) {
678                __setup_APIC_LVTT(calibration_result/prof_counter[cpu]);
679                prof_old_multiplier[cpu] = prof_counter[cpu];
680            }
681
682    #ifdef CONFIG_SMP
683            update_process_times(user);
684    #endif
685        }
686
687        /*
```

```
688          * We take the 'long' return path, and there every subsystem
689          * grabs the apropriate locks (kernel lock/ irq lock).
690          *
691          * we might want to decouple profiling from the 'long path',
692          * and do the profiling totally in assembly.
693          *
694          * Currently this isn't too much of an issue (performance wise),
695          * we can take more than 100K local irqs per second on a 100 MHz P5.
696          */
697     }
```

此外，读者在第 3 章中阅读过一些与时钟中断有关的代码，其中有些条件编译也是与 SMP 结构或内部 APIC 有关的，不妨回过去看一下。

9.5 SMP 结构中的进程调度

在单 CPU 的系统中，每一个给定的时刻只有当前进程是在运行中，其他所有的进程都不在运行。可是，在 SMP 结构的系统中却同时有好几个进程在运行，所以需要在进程的 task_struct 数据结构中加上两个字段。一个是 has_cpu，为 1 时表示进程正在某个 CPU 上运行，为 0 时则表示进程不在运行。另一个字段是 processor，当 has_cpu 为 1 时这个字段说明进程在哪一个 CPU 上运行。可想而知，一个进程只有在 has_cpu 字段为 0 才可以接受调度进入运行，这一点从宏操作 can_schedule( )的定义可以看出(kernel/sched.c)：

```
108     #ifdef CONFIG_SMP
109
110     #define idle_task(cpu) (init_tasks[cpu_number_map(cpu)])
111     #define can_schedule(p,cpu) ((!(p)->has_cpu) && \
112                     ((p)->cpus_allowed & (1 << cpu)))
113
114     #else
115
116     #define idle_task(cpu) (&init_task)
117     #define can_schedule(p,cpu) (1)
118
119     #endif
```

这里的 cpus_allowed 是 task_struct 数据结构中的另一个字段，它是一个位图。位图中的某一位为 1 就表示允许这个进程接受调度在相应的 CPU 上运行。从这个意义上讲，进程调度是个双向选择的过程。

当一个 CPU 通过 schedule( )从系统的就绪队列中挑选了一个进程作为运行的下一个进程 next，即将从当前进程 prev 切换到这个进程时，就将其 task_struct 结构中的 has_cpu 字段设置成 1，并将 processor 设置成该 CPU 的逻辑编号。我们回过头去（第 4 章）看一下 schedule( )中有关的片段(kernel/sched.c)：

```
508     asmlinkage void schedule(void)
509     {
```

```
        . . . . . . .
518         this_cpu = prev->processor;
        . . . . . . .
587     #ifdef CONFIG_SMP
588         next->has_cpu = 1;
589         next->processor = this_cpu;
590     #endif
591         spin_unlock_irq(&runqueue_lock);
592
593         if (prev == next)
594             goto same_process;
595
        . . . . . . .
648         switch_to(prev, next, prev);
649         __schedule_tail(prev);
        . . . . . . .
690     }
```

从进程 prev 切换到 next 以后，要对 prev 调用一个函数__schedule_tail( )。这个函数的代码对于 SMP 结构和单 CPU 结构有较大的不同（见 kernel/sched.c）。

[schedule( ) > __schedule_tail( )]

```
426     static inline void __schedule_tail(struct task_struct *prev)
427     {
428     #ifdef CONFIG_SMP
429         int policy;
430
431         /*
432          * prev->policy can be written from here only before `prev'
433          * can be scheduled (before setting prev >has_cpu to zero).
434          * Of course it must also be read before allowing prev
435          * to be rescheduled, but since the write depends on the read
436          * to complete, wmb( ) is enough. (the spin_lock( ) acquired
437          * before setting has_cpu is not enough because the spin_lock( )
438          * common code semantics allows code outside the critical section
439          * to enter inside the critical section)
440          */
441         policy = prev->policy;
442         prev->policy = policy & ~SCHED_YIELD;
443         wmb( );
444
445         /*
446          * fast path falls through. We have to clear has_cpu before
447          * checking prev->state to avoid a wakeup race - thus we
448          * also have to protect against the task exiting early.
449          */
```

```
450         task_lock(prev);
451         prev->has_cpu = 0;
452         mb( );
453         if (prev->state == TASK_RUNNING)
454             goto needs_resched;
455 
456     out_unlock:
457         task_unlock(prev);  /* Synchronise here with release_task( )
                                              if prev is TASK_ZOMBIE */
458         return;
459 
460         /*
461          * Slow path - we 'push' the previous process and
462          * reschedule_idle( ) will attempt to find a new
463          * processor for it. (but it might preempt the
464          * current process as well.) We must take the runqueue
465          * lock and re-check prev->state to be correct. It might
466          * still happen that this process has a preemption
467          * 'in progress' already - but this is not a problem and
468          * might happen in other circumstances as well.
469          */
470     needs_resched:
471         {
472             unsigned long flags;
473 
474             /*
475              * Avoid taking the runqueue lock in cases where
476              * no preemption-check is necessery:
477              */
478             if ((prev == idle_task(smp_processor_id( ))) ||
479                             (policy & SCHED_YIELD))
480                 goto out_unlock;
481 
482             spin_lock_irqsave(&runqueue_lock, flags);
483             if (prev->state == TASK_RUNNING)
484                 reschedule_idle(prev);
485             spin_unlock_irqrestore(&runqueue_lock, flags);
486             goto out_unlock;
487         }
488     #else
489         prev->policy &= ~SCHED_YIELD;
490     #endif /* CONFIG_SMP */
491     }
```

从代码中可见，这个 inline 函数对于单 CPU 系统只有一行，那就是 489 行，将原来的当前进程 prev 的 SCHED_YIELD 标志位设成 0（礼让只是一次有效），而这只在 prev 通过系统调用自愿礼让，暂时放弃运行时才有实际的作用。可是，对于 SMP 结构就不同了，除了也要把进程 prev 的 SCHED_YIELD

标志位设成 0 以外，这里还有几件事要做。首先是要把进程 prev 的 has_cpu 标志设成 0，表示这个进程已不在任何 CPU 上运行。代码中的 443 行和 452 行分别在写操作以后和读操作之前设置了内存路障，目的是：一方面要使其他 CPU 立即就能看到所作的改变，另一方面是要读到内存中最新的内容。另一件事是：如果进程 prev 的运行是被剥夺的，那就替它找找出路，看是否能让它继续在另一个 CPU 上运行。进程 prev 的状态为 TASK_RUNNING，说明它的运行是被剥夺的，或者是通过系统调用自愿礼让而暂时放弃的，所以，如果在 479 行排除了礼让的可能性，就必然是被剥夺了运行。不过，被剥夺运行的进程并不一定有必要，或者不一定可以为它另找一个处理器。如果这个进程是“空转”进程，也就是当前 CPU 上的 init 进程（参阅 SMP 结构的引导过程一节）的话，那一方面本来就无事可做，另一方面也根本不允许转到其他 CPU 上运行。把这些情况都排除了以后，就只剩下了一种可能，那就是这个进程确是有事可干，只是因为强制性调度被剥夺了运行。所以，此时要调用 reschedule_idle( )，尝试能否将其转到其他 CPU 上运行。这个函数的代码在 kernel/sched.c 中：

[schedule( ) > __schedule_tail( ) > reschedule_idle( )]

```
205     static void reschedule_idle(struct task_struct * p)
206     {
207     #ifdef CONFIG_SMP
208         int this cpu = smp_processor_id( );
209         struct task struct *tsk, *target_tsk;
210         int cpu, best cpu, i, max_prio;
211         cycles_t oldest idle;
212
213         /*
214          * shortcut if the woken up task's last CPU is
215          * idle now.
216          */
217         best_cpu = p->processor;
218         if (can_schedule(p, best cpu)) {
219             tsk = idle_task(best cpu);
220             if (cpu_curr(best_cpu) == tsk) {
221                 int need resched;
222     send_now_idle:
223                 /*
224                  * If need_resched == -1 then we can skip sending
225                  * the IPI altogether, tsk->need_resched is
226                  * actively watched by the idle thread.
227                  */
228                 need_resched = tsk->need_resched;
229                 tsk->need_resched = 1;
230                 if ((best_cpu != this_cpu) && !need_resched)
231                     smp_send_reschedule(best cpu);
232                 return;
233             }
234         }
235
```

```
236         /*
237          * We know that the preferred CPU has a cache-affine current
238          * process, lets try to find a new idle CPU for the woken-up
239          * process. Select the least recently active idle CPU. (that
240          * one will have the least active cache context.) Also find
241          * the executing process which has the least priority.
242          */
243         oldest_idle = (cycles_t) -1;
244         target_tsk = NULL;
245         max_prio = 1;
246 
247         for (i = 0; i < smp_num_cpus; i++) {
248             cpu = cpu_logical_map(i);
249             if (!can_schedule(p, cpu))
250                 continue;
251             tsk = cpu_curr(cpu);
252             /*
253              * We use the first available idle CPU. This creates
254              * a priority list between idle CPUs, but this is not
255              * a problem.
256              */
257             if (tsk == idle_task(cpu)) {
258                 if (last_schedule(cpu) < oldest_idle) {
259                     oldest_idle = last_schedule(cpu);
260                     target_tsk = tsk;
261                 }
262             } else {
263                 if (oldest_idle == -1ULL) {
264                     int prio = preemption_goodness(tsk, p, cpu);
265 
266                     if (prio > max_prio) {
267                         max_prio = prio;
268                         target_tsk = tsk;
269                     }
270                 }
271             }
272         }
273         tsk = target_tsk;
274         if (tsk) {
275             if (oldest_idle != -1ULL) {
276                 best_cpu = tsk->processor;
277                 goto send_now_idle;
278             }
279             tsk->need_resched = 1;
280             if (tsk->processor != this_cpu)
281                 smp_send_reschedule(tsk->processor);
282         }
283         return;
```

```
284
285
286    #else /* UP */
287        int this_cpu = smp_processor_id( );
288        struct task_struct *tsk;
289
290        tsk = cpu_curr(this_cpu);
291        if (preemption_goodness(tsk, p, this_cpu) > 1)
292            tsk->need_resched = 1;
293    #endif
294    }
```

代码中的 208～283 行用于 SMP 结构。这段代码先检查被剥夺运行的进程是否可以在原来的 CPU 上恢复运行（如果这个 CPU 上的当前进程是“空转”进程的话）。不行就进一步通过一个 for 循环依次考察系统中的所有 CPU，如果被剥夺运行的进程允许在某个 CPU 上运行，而这个 CPU 上的当前进程是“空转”进程，或者其“goodness”，即运行资格低于被剥夺运行的进程，那么这个 CPU 上的当前进程就是一个可以剥夺的候选对象。当系统中存在多个可以剥夺的进程时，就从中找出已经运行时间最长或运行资格最低的进程，总之是“大鱼吃小鱼”，“柿子拣软的捏”。到 for 循环结束时，如果找到了可以剥夺的进程，就将其 task_struct 结构中的 need_resched 标志设成 1，再向其所在的 CPU 发出一个 RESCHEDULE_VECTOR 中断请求；否则就只好算了。

信号的发送也与进程调度有关。在将一个信号发送给一个进程以后，要通过 signal_wake_up( )把目标进程唤醒。在单 CPU 的系统中，如果目标进程正在运行，那就必定是同一 CPU 上的当前进程，当 CPU 从中断或系统调用返回用户空间的前夕，就会处理这个信号。否则，目标进程也许是就绪进程；也许是个正在睡眠的进程，那就要将其唤醒。总之，到目标进程下一次受调度运行时就会先处理接收到的信号。在 SMP 结构中的情况要略为复杂一些，因为目标进程有可能正在另一个 CPU 上运行。为了使信号及时得到处理，此时要向正在执行目标进程的 CPU 发送一个 RESCHEDULE_VECTOR 中断请求。下面是有关的代码(kernel/signal.c)：

```
466    static inline void signal_wake_up(struct task_struct *t)
467    {
468        t->sigpending = 1;
469
470        if (t->state & TASK_INTERRUPTIBLE) {
471            wake_up_process(t);
472            return;
473        }
474
475    #ifdef CONFIG_SMP
476        /*
477         * If the task is running on a different CPU
478         * force a reschedule on the other CPU to make
479         * it notice the new signal quickly.
480         *
481         * The code below is a tad loose and might occasionally
```

```
482          * kick the wrong CPU if we catch the process in the
483          * process of changing - but no harm is done by that
484          * other than doing an extra (lightweight) IPI interrupt.
485          */
486         spin_lock(&runqueue_lock);
487         if (t->has_cpu && t->processor != smp_processor_id( ))
488             smp_send_reschedule(t->processor);
489         spin_unlock(&runqueue_lock);
490     #endif /* CONFIG_SMP */
491     }
```

这里的第 488 行向对方发出一个 RESCHEDULE_VECTOR 中断请求。中断请求的直接目的并不在于让对方进行一次进程调度，而只是使对方产生一次中断，为及时处理刚投递的信号创造条件。

9.6 SMP 系统的引导

SMP 结构中的所有 CPU 都是平等的，没有主次之分，这实际上是建立在系统中有多个进程或者多个执行“上下文”前提下的。在同一时间中，一个“上下文”只能由一个 CPU 处理，否则只会把事情搞糟。如果系统中一共才只有一个“上下文”，那么有再多的 CPU 存在也无从发挥作用。所以，系统的引导和初始化阶段是个特例，因为在这个阶段里系统中只有一个“上下文”，只能由一个处理器来处理。在这个阶段里，也就是在系统刚加电或“总清”(reset)之后，系统中暂时只有一个处理器运行，这个处理器称为“引导处理器”BP；其余的处理器则处于暂停状态，称为“应用处理器”AP。“引导处理器” 完成了系统的引导和初始化，并创建起多个进程，从而可以由多个处理器同时参与处理时，才启动所有的“应用处理器”，让它们在完成自身的初始化以后投入运行。一旦各个“应用处理器”都已投入运行，这种暂时的主次关系便告结束，从此以后便一律平等了。系统的引导和初始化本是下一章的题材，但是我们在这里关心的是“引导处理器”怎样为各个“应用处理器”作好运行的准备，然后启动其运行的过程。这个过程固然是整个系统初始化过程中的一部分，但是实际上与 SMP 结构的关系更为密切，所以放在本章中叙述。

在初始化阶段，“引导处理器”先完成自身的初始化，进入保护模式并开启页式存储管理机制，再完成系统特别是内存的初始化，然后就从 start_kernel( )调用 smp_init( )进行 SMP 结构的初始化。这个函数的代码在 init/main.c 中：

[start_kernel( ) > smp_init( )]

```
505     /* Called by boot processor to activate the rest. */
506     static void __init smp_init(void)
507     {
508         /* Get other processors into their bootup holding patterns. */
509         smp_boot_cpus( );
510         smp_threads_ready=1;
511         smp_commence( );
512     }
```

这个函数的主体是 smp_boot_cpus( )，它依次启动系统中的各个 CPU，让它们各自走过初始化的第一阶段。各个次 CPU 在完成了自身的初始化以后都要停下来等待一个统一的“起跑”命令。而主 CPU，则在完成了所有次 CPU 的启动以后通过 smp_commence( )发出这个命令。我们先看 smp_boot_cpus( )，其代码在 rch/i386/kernel/smpboot.c 中。我们分段阅读。

[start_kernel( ) > smp_init( ) > smp_boot_cpus( )]

```
829     void __init smp_boot_cpus(void)
830     {
831         int apicid, cpu;
832
833     #ifdef CONFIG_MTRR
834         /*  Must be done before other processors booted  */
835         mtrr_init_boot_cpu ( );
836     #endif
837         /*
838          * Initialize the logical to physical CPU number mapping
839          * and the per-CPU profiling counter/multiplier
840          */
841
842         for (apicid = 0; apicid < NR_CPUS; apicid++) {
843             x86_apicid_to_cpu[apicid] = -1;
844             prof_counter[apicid] = 1;
845             prof_old_multiplier[apicid] = 1;
846             prof_multiplier[apicid] = 1;
847         }
848
849         /*
850          * Setup boot CPU information
851          */
852         smp_store_cpu_info(0); /* Final full version of the data */
853         printk("CPU%d: ", 0);
854         print_cpu_info(&cpu_data[0]);
855
856         /*
857          * We have the boot CPU online for sure.
858          */
859         set_bit(0, &cpu_online_map);
860         x86_apicid_to_cpu[boot_cpu id] = 0;
861         x86_cpu_to_apicid[0] = boot cpu_id;
862         global_irq_holder = 0;
863         current->processor = 0;
864         init_idle( );
865         smp_tune_scheduling( );
866
867         /*
868          * If we couldnt find an SMP configuration at boot time,
```

```
869          * get out of here now!
870          */
871         if (!smp found_config) {
872             printk(KERN_NOTICE
                     "SMP motherboard not detected. Using dummy APIC emulation.\n");
873     #ifndef CONFIG_VISWS
874             io_apic_irqs = 0;
875     #endif
876             cpu_online_map = phys_cpu_present_map = 1;
877             smp_num_cpus = 1;
878             goto smp_done;
879         }
880
```

先是一些准备工作。前面提到过，对内存的高速缓冲可以通过一个“存储类型及范围寄存器”，即 MTRR 加以分区管理。例如某一个区间采用“穿透”写模式，而另一个区间采用“回写”模式，再另一个区间则根本就不缓冲，等等。MTRR 并不是非用不可的，它的使用只是使管理更加精细而已，所以是否采用 MTRR 是个编译选择项。但是，如果选择了使用 MTRR，就要在启动次 CPU 之前先通过 mtrr_init_boot_cpu( )完成对主 CPU 的 MTRR 的初始化。此外，内核中还有一些以 CPU 编号为下标的数组，其中 x86_apicid_to_cpu[ ]用于逻辑 CPU 号到物理 CPU 号的转换，还有些是为统计信息而设的，对这些数组都要加以初始化。然后，还要调用 smp_store_cpu_info( )，从 CPU 收集很多信息，并根据收集的信息进行一些必要的操作。这些信息大多是通过指令 cpuid 收集的，通过这条指令可以收集到许多有关 CPU 本身的信息。例如，这些信息中不但包括 CPU 的型号，还包括由哪一家厂商制造。如果发现 CPU 是由 AMD 制造的，就要相应地调用一个针对 AMD 处理器特点的初始化函数。这些信息存贮在一个 cpuinfo_x86 结构数组 cpu_data[ ]中，这个数组中的内容可以通过特殊文件系统/proc 读取。在 /proc 目录下有个文件/proc/cpuinfo，对这个特殊文件的读操作就是从 cpu_data[ ]中读出的。读者不妨试一下“more /proc/cpuinfo”，看看你的机器用的是什么 CPU。

主 CPU 的逻辑号总是 0，而物理号则在一个全局量 boot_cpu_id 中，数组 x86_apicid_to_cpu[ ]和 x86_cpu_to_apicid[ ]提供了二者间的转换。内核中还有个全局量 smp_found_config，由主 CPU 在一个函数 setup_arch( )中调用 find_smp_config( )，根据 BIOS 提供的信息设置，为 0 表示系统中只有一个 CPU，否则就是多 CPU 系统。所以，如果此时 smp_found_config 为 0，就结束了 smp_boot_cpus( )的执行；否则就继续往下执行。

[start_kernel( ) > smp_init( ) > smp_boot_cpus( )]

```
881         /*
882          * Should not be necessary because the MP table should list the boot
883          * CPU too, but we do it for the sake of robustness anyway.
884          */
885         if (!test bit(boot_cpu_id, &phys_cpu_present_map)) {
886             printk("weird, boot CPU (#%d) not listed by the BIOS.\n",
887                                         boot_cpu_id);
888             phys_cpu_present_map |= (1 << hard_smp_processor_id( ));
889         }
```

```
890
891         /*
892          * If we couldn't find a local APIC, then get out of here now!
893          */
894         if (APIC_INTEGRATED(apic_version[boot_cpu_id]) &&
895             !test_bit(X86_FEATURE_APIC, boot_cpu_data.x86_capability)) {
896             printk(KERN_ERR "BIOS bug, local APIC #%d not detected!...\n",
897                 boot_cpu_id);
898           printk(KERN_ERR
              "... forcing use of dummy APIC emulation. (tell your hw vendor)\n");
899     #ifndef CONFIG_VISWS
900             io_apic_irqs = 0;
901     #endif
902             cpu_online_map = phys_cpu_present_map = 1;
903             smp_num_cpus = 1;
904             goto smp_done;
905         }
906
907         verify_local_APIC( );
908
909         /*
910          * If SMP should be disabled, then really disable it!
911          */
912         if (!max_cpus) {
913             smp_found_config = 0;
914           printk(KERN_INFO
                "SMP mode deactivated, forcing use of dummy APIC emulation.\n");
915     #ifndef CONFIG_VISWS
916             io_apic_irqs = 0;
917     #endif
918             cpu_online_map = phys_cpu_present_map = 1;
919             smp_num_cpus = 1;
920             goto smp_done;
921         }
922
923         connect_bsp_APIC( );
924         setup_local_APIC( );
925
926         if (GET_APIC_ID(apic_read(APIC_ID)) != boot_cpu_id)
927             BUG( );
928
```

这里还有一系列检验。其中第 894 行检查主 CPU 是否带有内部 APIC。对于 SMP 结构，CPU 带有内部 APIC 是个必要条件。随后，又对主 CPU 的内部 APIC 进行了初始化。限于篇幅，我们不能对调用的子程序一一加以说明了，有需要的读者可以结合 Intel 的技术资料自己阅读。这里的 phys_cpu_present_map 是个全局的 CPU 位图，由主 CPU 在 setup_arch( )（见第 10 章）中通过 get_smp_config( )辗转调用 MP_processor_info( )，根据 BIOS 提供的信息设置。

至此，所有的准备工作都已完成，下面就要逐个地启动系统中的次 CPU 了。

[start_kernel( ) > smp_init( ) > smp_boot_cpus( )]

```
929         /*
930          * Now scan the CPU present map and fire up the other CPUs.
931          */
932         Dprintk("CPU present map: %lx\n", phys_cpu_present_map);
933 
934         for (apicid = 0; apicid < NR_CPUS; apicid++) {
935             /*
936              * Don't even attempt to start the boot CPU!
937              */
938             if (apicid == boot_cpu_id)
939                 continue;
940 
941             if (!(phys_cpu_present_map & (1 << apicid)))
942                 continue;
943             if ((max_cpus >= 0) && (max_cpus <= cpucount+1))
944                 continue;
945 
946             do_boot_cpu(apicid);
947 
948             /*
949              * Make sure we unmap all failed CPUs
950              */
951             if ((x86_apicid_to_cpu[apicid] == -1) &&
952                     (phys_cpu_present_map & (1 << apicid)))
953                 printk("phys CPU #%d not responding - cannot use it.\n",
                                                                  apicid);
954         }
955 
```

内核中有个全局量 max_cpus，表示系统中有多少个 CPU 其值就是多少，但是也可以在系统引导命令行中指定只用其中几个。此外，这里的 cpucount 是个计数器，初值为 0。代码中的 for 循环根据位图 phys_cpu_present_map 依次对各个“应用处理器”调用 do_boot_cpu( )，为其投入运行作好准备，并启动其运行。这个函数的代码在 arch/i386/kernel/smp.c 中。由于比较长，我们也只好分段阅读。

[start_kernel( ) > smp_init( ) > smp_boot_cpus( ) > do_boot_cpu( )]

```
541     static void __init do_boot_cpu (int apicid)
542     {
543         struct task_struct *idle;
544         unsigned long send_status, accept_status, boot_status, maxlvt;
545         int timeout, num_starts, j, cpu;
546         unsigned long start_eip;
547 
```

```
548         cpu = ++cpucount;
549         /*
550          * We can't use kernel_thread since we must avoid to
551          * reschedule the child.
552          */
553         if (fork_by_hand( ) < 0)
554             panic("failed fork for CPU %d", cpu);
555 
556         /*
557          * We remove it from the pidhash and the runqueue
558          * once we got the process:
559          */
560         idle = init_task.prev_task;
561         if (!idle)
562             panic("No idle process for CPU %d", cpu);
563 
564         idle->processor = cpu;
565         x86_cpu_to_apicid[cpu] = apicid;
566         x86_apicid_to_cpu[apicid] = cpu;
567         idle->has_cpu = 1; /* we schedule the first task manually */
568         idle->thread.eip = (unsigned long) start_secondary;
569 
570         del_from_runqueue(idle);
571         unhash_process(idle);
572         init_tasks[cpu] = idle;
573 
574         /* start_eip had better be page aligned! */
575         start_eip = setup_trampoline( );
576 
577         /* So we see what's up   */
578         printk("Booting processor %d/%d eip %lx\n", cpu, apicid, start_eip);
579         stack_start.esp = (void *) (1024 + PAGE_SIZE + (char *)idle);
580 
```

前面讲过，每个CPU在运行中必须有自己的上下文，否则就不会对整个系统的运行作出贡献、反而倒会造成损害。所以，必须为每个CPU都准备下一个初始进程（线程）。那怕这个初始进程本身实际上并没有什么事要做，也得要有这么一个进程才能启动这个CPU。而CPU一旦开始了初始进程的运行，以后就可以通过进程调度从系统中挑选其他进程运行了。所以，第一件要做的事就是通过fork_by_hand( )为目标CPU创建起一个内核线程。

[start_kernel( ) > smp_init( ) > smp_boot_cpus( ) > do_boot_cpu( ) > fork_by_hand( )]

```
493     static int __init fork_by_hand(void)
494     {
495         struct pt_regs regs;
496         /*
497          * don't care about the eip and regs settings since
```

```
498         * we'll never reschedule the forked task.
499         */
500        return do_fork(CLONE_VM|CLONE_PID, 0, &regs, 0);
501    }
```

读者对 do_fork( )已经不陌生了。这里的第一个参数表示 fork 的是一个线程，共享父进程，即当前进程的用户空间，并且还共享父进程的进程号。第二个参数本来应该是用户空间堆栈的起始地址，第四个参数为堆栈的大小，但是因为创建的是内核线程，所以这两个参数都是 0。寄存器组 regs 的具体内容在这里并没有实际的作用。创建了所需的内核线程以后，do_fork( )通过 SET_LINKS( )，把它的 task_struct 数据结构链接在就绪进程队列中 init_task 的前面，所以第 560 行可以通过 init_task 取得指向这个 task_struct 数据结构的指针 idle。

代码中的 568 行将 idle->thread.eip 设置成指向 start_secondary( )，这是所有的次 CPU，即“应用处理器”，完成了初始化以后进入正常调度时的入口。后面我们会看到它的代码。然后，del_from_runqueue( )和 unhash_process( )将这个 task_struct 结构从就绪进程队列和杂凑队列中删去，并把它的指针放在数组 init_tasks[ ]中。这样就只能根据 CPU 序号找到这个 task_struct 结构，而不能像常规的进程那样通过进程号找到它了。读者后面会看到，这个线程就是给定 CPU 的“空转”进程。所有 CPU 的空转进程都具有相同的进程号 0，并且都不挂入就绪进程队列和杂凑队列内，而由数组 init_tasks[ ]的各个元素指向相应的 task_struct 结构。

函数 start_secondary( )的执行有个前提，那就是次 CPU 已经完成了其本身的初始化，已进入了保护模式，并开启了页式存储管理。可是，次 CPU 本身的这些初始化涉及其内部寄存器的操作，因而只能由它自己完成，主 CPU 无法包办代替。再说，当主 CPU 启动一个次 CPU 运行的时候，这个次 CPU 一开始时还处于实地址模式，根本就不能正确地执行 start_secondary( )的代码。因此，start_secondary( )的作用其实还只是中转性质的，次 CPU 仍然不能“一步登天”进入这个函数中执行，而要先进入一个初始化程序 startup_32( )，完成 CPU 本身的初始化（见第 10 章）。可是，即使 startup_32( )的执行也还有前提，所以次 CPU 在受到启动之初甚至还不能直接进入初始化的第一阶段 startup_32( )，而需要有块“跳板”再来一次中转。读者在第 10 章中将会看到，主 CPU 在进入 startup_32( )之前已经在引导辅助程序中进行了一些准备，包括进入保护模式；次 CPU 同样需要这些准备。此外，次 CPU 在进入 startup_32( )之前应该把%ebx 的内容设置成 1（而主 CPU 则为 0）。显然，这些准备工作都要在进入 startup_32( )之前完成。这就是为什么次 CPU 需要“跳板”而主 CPU 却并不需要的原因。这里 575 行通过 setup_trampoline( )为次 CPU 复制好一块跳板，实际上是一段汇编语言程序，并把这段程序的入口地址放在变量 start_eip 中。所以，start_eip 中的地址是启动次 CPU 时的第一站，是低级阶段。而 idle->thread.eip 中的地址则是次 CPU 在完成了自身的初始化，建立起了页面映射以后才开始执行的新起点，是第二站，是高级阶段。函数 setup_trampoline( )的代码在 arch/i386/kernel/smpboot.c 中：

[start_kernel( ) > smp_init( ) > smp_boot_cpus( ) > do_boot_cpu( ) > setup_trampoline( )]

```
101    /*
102     * Trampoline 80x86 program as an array.
103     */
104
105    extern unsigned char trampoline_data [ ];
106    extern unsigned char trampoline_end  [ ];
```

```
107    static unsigned char *trampoline base;
108
109    /*
110     * Currently trivial. Write the real->protected mode
111     * bootstrap into the page concerned. The caller
112     * has made sure it's suitably aligned.
113     */
114
115    static unsigned long __init setup_trampoline(void)
116    {
117      memcpy(trampoline base, trampoline_data, trampoline end - trampoline_data);
118        return virt_to phys(trampoline_base);
119    }
```

这段程序是再简单不过的了，但是读者一定会对这“跳板”到底是什么感兴趣。这段代码在 arch/i386/kernel/trampoline.S 中（trampoline 就是跳板的意思）：

```
37     ENTRY(trampoline_data)
38     r_base = .
39
40         mov %cs, %ax                # Code and data in the same place
41         mov %ax, %ds
42
43         mov $1, %bx                 # Flag an SMP trampoline
44         cli                         # We should be safe anyway
45
46         movl    $0xA5A5A5A5, trampoline_data - r_base
47                                     # write marker for master knows we're running
48
49         lidt    idt_48 - r_base     # load idt with 0, 0
50         lgdt    gdt_48 - r_base     # load gdt with whatever is appropriate
51
52         xor %ax, %ax
53         inc %ax                     # protected mode (PE) bit
54         lmsw    %ax                 # into protected mode
55         jmp flush instr
56     flush_instr:
57         ljmpl       $__KERNEL_CS, $0x00100000
58                                     # jump to startup_32 in arch/i386/kernel/head.S
59
60     idt_48:
61         .word       0               # idt limit = 0
62         .word       0, 0            # idt base = 0L
63
64     gdt_48:
65         .word       0x0800          # gdt limit = 2048, 256 GDT entries
66         .long       gdt_table-__PAGE_OFFSET # gdt base = gdt (first SMP CPU)
```

```
67
68    .globl SYMBOL_NAME(trampoline end)
```

我们在这里不深入解读这段程序了，读者不妨自己加以研究。我们只指出三点：第一，43 行把寄存器%bx 的内容设置成 1，表示这是一个次 CPU。第二，52～54 行将控制寄存器 CR0 的内容设置成 1，就是把 CR0 中的 PE 位(最低位，为 1 表示保护模式)设置成 1，其余位则全都为 0，这样就使 CPU 进入了保护模式，但是页式存储管理则尚未启用(最高位 PG 为 0)。第三，57 行跳转到代码段__KERNEL_CS 中地址为 0x00100000 的地方。读者在第 2 章中看到，常数__KERNEL_CS 的值定义为 0x10，而段描述表 gdt_48 中与此相应的段描述项提供的基地址为 0，所以跳转的目标是 0x00100000。这就是 1MB 处，就是系统在引导以后初始化程序的起点 startup_32( )。那正是下一章的重点之一。

读者可能还有个问题：为什么要把这段代码复制到 trampoline_base 中呢？说来话长。当主 CPU 通过 APIC 启动次 CPU 运行时，要把一个启动地址发送给次 CPU。可是，由于 APIC 的内部结构，实际上能发送的只是一个 8 位的物理页面号，称为“向量”。这样，就给启动地址加上了限制：首先，必须与页面边界对齐；其次，必须在 1MB 以下。显然，这里的 trampoline_data( )是不满足这两个条件的，所以要另行在 1MB 以下分配一个物理页面，把 trampoline_data( )的代码复制到这个页面中，这就是 trampoline_base，是在主 CPU 的初始化阶段通过一个函数 smp_alloc_memory( )分配的。

回到 do_boot_cpu( )的代码中，579 行把次 CPU 执行 start_secondary( )时的堆栈，设置在其 task_struct 数据结构所在的两个页面中(详见第 3 章)。我们继续往下阅读。

[start_kernel( ) > smp_init( ) > smp_boot_cpus( ) > do_boot_cpu( )]

```
581         /*
582          * This grunge runs the startup process for
583          * the targeted processor.
584          */
585
586         atomic set(&init_deasserted, 0);
587
588         Dprintk("Setting warm reset code and vector.\n");
589
590         CMOS_WRITE(0xa, 0xf);
591         local_flush_tlb( );
592         Dprintk("1.\n");
593         *((volatile unsigned short *) phys_to_virt(0x469)) = start eip >> 4;
594         Dprintk("2.\n");
595         *((volatile unsigned short *) phys_to_virt(0x467)) = start_eip & 0xf;
596         Dprintk("3.\n");
597
```

主 CPU 在启动次 CPU 之前，先要通过其本地 APIC 对次 CPU 的 APIC 执行一次初始化操作，在此期间还把一个全局量 init_deasserted 设成 0，到完成了对目标 APIC 的初始化以后，再把这个变量设置成 1。此外，上述 start_eip 中的入口地址不仅用于本次初始化，还用于系统的“热启动”，所以按照规定将其写入物理地址为 0x467 和 0x469 处。

现在，主 CPU 可以执行给定次 CPU 的启动了。限于篇幅，我们在下面将不深入考察对 APIC 寄存

器的操作，有兴趣或有需要的读者可自己加以研究。总而言之，这段代码通过主CPU的本地APIC中的一些寄存器，包括用来说明发送目标的寄存器APIC_ICR2和控制/状态寄存器APIC_ICR，向目标CPU的APIC发出一些信号，并等待其完成。

[start_kernel( ) > smp_init( ) > smp_boot_cpus( ) > do_boot_cpu( )]

```
598         /*
599          * Be paranoid about clearing APIC errors.
600          */
601         if (APIC_INTEGRATED(apic_version[apicid])) {
602             apic_read_around(APIC_SPIV);
603             apic_write(APIC_ESR, 0);
604             apic_read(APIC_ESR);
605         }
606 
607         /*
608          * Status is now clean
609          */
610         send_status = 0;
611         accept_status = 0;
612         boot_status = 0;
613 
614         /*
615          * Starting actual IPI sequence...
616          */
617 
618         Dprintk("Asserting INIT.\n");
619 
620         /*
621          * Turn INIT on target chip
622          */
623         apic_write_around(APIC_ICR2, SET_APIC_DEST_FIELD(apicid));
624 
625         /*
626          * Send IPI
627          */
628         apic_write_around(APIC_ICR, APIC_INT_LEVELTRIG | APIC_INT_ASSERT
629                             | APIC_DM_INIT);
630 
631         Dprintk("Waiting for send to finish...\n");
632         timeout = 0;
633         do {
634             Dprintk("+");
635             udelay(100);
636             send_status = apic_read(APIC_ICR) & APIC_ICR_BUSY;
637         } while (send_status && (timeout++ < 1000));
638 
```

```
639         mdelay(10);
640 
641         Dprintk("Deasserting INIT.\n");
642 
643         /* Target chip */
644         apic_write_around(APIC_ICR2, SET_APIC_DEST_FIELD(apicid));
645 
646         /* Send IPI */
647         apic_write_around(APIC_ICR, APIC_INT_LEVELTRIG | APIC_DM_INIT);
648 
649         Dprintk("Waiting for send to finish...\n");
650         timeout = 0;
651         do {
652             Dprintk("+");
653             udelay(100);
654             send_status = apic_read(APIC_ICR) & APIC_ICR_BUSY;
655         } while (send_status && (timeout++ < 1000));
656 
657         atomic_set(&init_deasserted, 1);
658 
```

发出的信号须符合一定的顺序，例如上面的 628 行中先把控制位 APIC_INT_ASSERT 设成 1，然后在下面的 647 行再把它设成 0。每次写入控制/状态寄存器 APIC_ICR 以后，都要从这个寄存器读回来，并等待其状态位 APIC_ICR_BUSY 变成 0。不过，等待也不能是无限期的等待，所以还要通过计数加以限制。

[start_kernel( ) > smp_init( ) > smp_boot_cpus( ) > do_boot_cpu( )]

```
659         /*
660          * Should we send STARTUP IPIs ?
661          *
662          * Determine this based on the APIC version.
663          * If we don't have an integrated APIC, don't
664          * send the STARTUP IPIs.
665          */
666         if (APIC_INTEGRATED(apic_version[apicid]))
667             num_starts = 2;
668         else
669             num_starts = 0;
670 
671         /*
672          * Run STARTUP IPI loop.
673          */
674         Dprintk("#startup loops: %d.\n", num_starts);
675 
676         maxlvt = get_maxlvt( );
677 
```

```
678         for (j = 1; j <= num_starts; j++) {
679             Dprintk("Sending STARTUP #%d.\n", j);
680             apic_read_around(APIC SPIV);
681             apic_write(APIC_ESR, 0);
682             apic_read(APIC_ESR);
683             Dprintk("After apic_write.\n");
684
685             /*
686              * STARTUP IPI
687              */
688
689             /* Target chip */
690             apic_write around(APIC_ICR2, SET_APIC_DEST_FIELD(apicid));
691
692             /* Boot on the stack */
693             /* Kick the second */
694             apic_write_around(APIC_ICR, APIC_DM_STARTUP
695                         | (start_eip >> 12));
696
697             /*
698              * Give the other CPU some time to accept the IPI.
699              */
700             udelay(300);
701
702             Dprintk("Startup point 1.\n");
703
704             Dprintk("Waiting for send to finish...\n");
705             timeout = 0;
706             do {
707                 Dprintk("+");
708                 udelay(100);
709                 send status = apic read(APIC_ICR) & APIC_ICR_BUSY;
710             } while (send_status && (timeout++ < 1000));
711
712             /*
713              * Give the other CPU some time to accept the IPI.
714              */
715             udelay(200);
716             /*
717              * Due to the Pentium erratum 3AP.
718              */
719             if (maxlvt > 3) {
720                 apic_read around(APIC_SPIV);
721                 apic_write(APIC_ESR, 0);
722             }
723             accept_status = (apic_read(APIC ESR) & 0xEF);
724             if (send_status || accept_status)
725                 break;
```

```
726         }
```

最后，将 start_eip 中的初始化程序入口地址发送给目标 CPU，并等待目标 CPU 的回应。注意这里的 694 和 695 行将 start_eip 中的启动地址写入 APIC 的控制寄存器 APIC_ICR。这个地址是物理地址，在写入寄存器前先右移了 12 位，因为启动地址一定是与页面边界对齐的，其低 12 位一定是 0。另一方面，启动地址一定在 1MB 以下，其最高 12 位也一定是 0，所以右移以后真正要发送出去的是 8 位。

[start_kernel( ) > smp_init( ) > smp_boot_cpus( ) > do_boot_cpu( )]

```
727         Dprintk("After Startup.\n");
728 
729         if (send_status)
730             printk("APIC never delivered???\n");
731         if (accept_status)
732             printk("APIC delivery error (%lx).\n", accept_status);
733 
734         if (!send_status && !accept_status) {
735             /*
736              * allow APs to start initializing.
737              */
738             Dprintk("Before Callout %d.\n", cpu);
739             set_bit(cpu, &cpu_callout_map);
740             Dprintk("After Callout %d.\n", cpu);
741 
742             /*
743              * Wait 5s total for a response
744              */
745             for (timeout = 0; timeout < 50000; timeout++) {
746                 if (test_bit(cpu, &cpu_callin_map))
747                     break;  /* It has booted */
748                 udelay(100);
749             }
750 
751             if (test_bit(cpu, &cpu_callin_map)) {
752                 /* number CPUs logically, starting from 1 (BSP is 0) */
753                 Dprintk("OK.\n");
754                 printk("CPU%d: ", cpu);
755                 print_cpu_info(&cpu_data[cpu]);
756                 Dprintk("CPU has booted.\n");
757             } else {
758                 boot_status = 1;
759                 if (*((volatile unsigned char *)phys_to_virt(8192))
760                         == 0xA5)
761                     /* trampoline started but...? */
762                     printk("Stuck ??\n");
763                 else
764                     /* trampoline code not run */
```

```
765                     printk("Not responding.\n");
766     #if APIC_DEBUG
767                 inquire_remote_apic(apicid);
768     #endif
769             }
770         }
771         if (send_status || accept_status || boot_status) {
772             x86_cpu_to_apicid[cpu] = -1;
773             x86_apicid_to_cpu[apicid] = -1;
774             cpucount--;
775         }
776
777         /* mark "stuck" area as not stuck */
778         *((volatile unsigned long *)phys_to_virt(8192)) = 0;
779     }
```

次 CPU 受到启动以后，首先进入上述的“跳板”，从那里起跳，进入 startup_32()，即初始化的第一阶段（详见第 10 章）。在 startup_32()中，由于次 CPU 的寄存器%ebx 中的内容为 1，就可以跟主 CPU 区分开来，因为有些操作是共同的，而又有些则只由主 CPU 完成。完成了 CPU 本身初始化以后，次 CPU 就通过函数调用进入 initialize_secondary()（见第 10 章，代码在 arch/i386/kernel/smpboot.c 中）：

[startup_32() > initialize_secondary()]

```
468     /*
469      * Everything has been set up for the secondary
470      * CPUs - they just need to reload everything
471      * from the task structure
472      * This function must not return.
473      */
474     void __init initialize_secondary(void)
475     {
476         /*
477          * We don't actually need to load the full TSS,
478          * basically just the stack pointer and the eip.
479          */
480
481         asm volatile(
482             "movl %0,%%esp\n\t"
483             "jmp *%1"
484             :
485             :"r" (current->thread.esp),"r" (current->thread.eip));
486     }
```

次 CPU 在 startup_32()中，将自己的堆栈设置在主 CPU 为其准备的地方，从而进入了自己的上下文，这就是主 CPU 为之准备好的空转进程。我们在前面看到，这个进程的 thread.eip 指向 start_secondary()，所以 483 行的 jmp 指令就使 CPU 进入了这个函数。同时，482 行已经重新设置了堆

栈指针，所以实际上永远不会从 initialize_secondary( )返回了。函数 start_secondary( )的代码在 arch/i386/kernel/smpboot.c 中：

```
445  /*
446   * Activate a secondary processor.
447   */
448  int __init start_secondary(void *unused)
449  {
450      /*
451       * Dont put anything before smp_callin( ), SMP
452       * booting is too fragile that we want to limit the
453       * things done here to the most necessary things.
454       */
455      cpu_init( );
456      smp_callin( );
457      while (!atomic_read(&smp_commenced))
458          rep_nop( );
459      /*
460       * low-memory mappings have been cleared, flush them from
461       * the local TLBs too.
462       */
463      local_flush_tlb( );
464
465      return cpu_idle( );
466  }
```

对次 CPU 的基本初始化是在 startup_32( )中完成的，但是进一步的初始化则放在进入 start_secondary( )以后才进行，那就是 cpu_init( )。这个函数不仅次 CPU 要调用，主 CPU 也要在初始化的第二阶段中调用，其代码可参看第 10 章中的“系统初始化（第二阶段）”一节。与 startup_32( )中的初始化相比，这个函数主要是为进程调度作准备。此外，在这个函数中还将全局的位图 cpu_initialized 中与当前 CPU 对应的标志位设成 1，让主 CPU 知道这个次 CPU 的初始化已经完成了。

执行完 cpu_init( )以后，次 CPU 的初始化就完成了。这里还要说明一下，我们现在是在讲述其中一个次 CPU 的运行，一个次 CPU 的上下文。主 CPU 在 smp_boot_cpus( )的一个 for 循环中，依次对系统中的各个次 CPU 调用 do_boot_cpu( )，启动其运行。每启动一个次 CPU，这个 CPU 就会进入 start_secondary( )。但是，主 CPU 和次 CPU 之间需要一些通信手段来建立起若干同步点，以取得互相的同步和协调。例如，主 CPU 需要确认刚被启动的次 CPU 已经到达 start_secondary( )以后，才能启动下一个次 CPU。而且，不仅主 CPU 和个别的次 CPU 之间需要同步，所有的次 CPU 最后也要步调一致，基本上在同一时间进入 cpu_idle( )。具体的同步有下面这么一些：

(1) 全局量 init_deasserted。在 do_boot_cpu( )中，主 CPU 在对次 CPU 的 APIC 进行初始化前，先将 init_deasserted 设成 0（586 行），完成了以后再将这个变量设成 1（657 行）。次 CPU 受到启动、通过 start_secondary( )进入 smp_callin( )以后，就在一个 while 循环中等待这个变量成为 1，从而保证次 CPU 不会在此期间进行对 APIC 的操作。

(2) 全局量位图 cpu_callout_map。次 CPU 在 smp_callin( )中还要等待这个位图中的对应位变成 1(最多等待 2 秒)。主 CPU 则在 do_boot_cpu( )中（739 行）将目标 CPU 的对应位设成 1。

(3) 全局量位图 cpu_callin_map。主 CPU 在 do_boot_cpu( )中向次 CPU 发出启动命令以后，就在一个定时的循环中测试目标 CPU 在这个位图中的对应位，等待来自目标 CPU 的回答(745～749 行)。次 CPU 则在 smp_callin( )结束之前将位图中的对应位设置成 1。主 CPU 只有测试到这一位变成 1，或者超过了预定的时间，才从 do_boot_cpu( )返回。

(4) 全局量 smp_commenced。所有次 CPU 在进入 cpu_init( )之前都要等待这个变量成为 1，这个变量将使所有的次 CPU 都站到了同一条起跑线上。

函数 smp_callin ( )的代码在 arch/i386/kernel/smpboot.c 中：

[start_secondary( ) > smp_callin( )]

```
349	void __init smp_callin(void)
350	{
351		int cpuid, phys_id;
352		unsigned long timeout;
353	
354		/*
355		 * If waken up by an INIT in an 82489DX configuration
356		 * we may get here before an INIT-deassert IPI reaches
357		 * our local APIC.  We have to wait for the IPI or we'll
358		 * lock up on an APIC access.
359		 */
360		while (!atomic_read(&init_deasserted));
361	
362		/*
363		 * (This works even if the APIC is not enabled.)
364		 */
365		phys_id = GET_APIC_ID(apic_read(APIC_ID));
366		cpuid = current->processor;
367		if (test_and_set_bit(cpuid, &cpu_online_map)) {
368			printk("huh, phys CPU#%d, CPU#%d already present??\n",
369					phys_id, cpuid);
370			BUG( );
371		}
372		Dprintk("CPU#%d (phys ID: %d) waiting for CALLOUT\n", cpuid, phys_id);
373	
374		/*
375		 * STARTUP IPIs are fragile beasts as they might sometimes
376		 * trigger some glue motherboard logic. Complete APIC bus
377		 * silence for 1 second, this overestimates the time the
378		 * boot CPU is spending to send the up to 2 STARTUP IPIs
379		 * by a factor of two. This should be enough.
380		 */
381	
382		/*
383		 * Waiting 2s total for startup (udelay is not yet working)
384		 */
```

```
385         timeout = jiffies + 2*HZ;
386         while (time_before(jiffies, timeout)) {
387             /*
388              * Has the boot CPU finished it's STARTUP sequence?
389              */
390             if (test_bit(cpuid, &cpu_callout_map))
391                 break;
392         }
393 
394         if (!time_before(jiffies, timeout)) {
395             printk("BUG: CPU%d started up but did not get a callout!\n",
396                 cpuid);
397             BUG( );
398         }
399 
400         /*
401          * the boot CPU has finished the init stage and is spinning
402          * on callin_map until we finish. We are free to set up this
403          * CPU, first the APIC. (this is probably redundant on most
404          * boards)
405          */
406 
407         Dprintk("CALLIN, before setup_local_APIC( ).\n");
408         setup_local_APIC( );
409 
410         sti( );
411 
412     #ifdef CONFIG_MTRR
413         /*
414          * Must be done before calibration delay is computed
415          */
416         mtrr_init_secondary_cpu ( );
417     #endif
418         /*
419          * Get our bogomips.
420          */
421         calibrate_delay( );
422         Dprintk("Stack at about %p\n",&cpuid);
423 
424         /*
425          * Save our processor parameters
426          */
427         smp_store_cpu_info(cpuid);
428 
429         /*
430          * Allow the master to continue.
431          */
432         set_bit(cpuid, &cpu_callin_map);
```

```
433
434         /*
435          *      Synchronize the TSC with the BP
436          */
437         if (cpu_has_tsc)
438             synchronize_tsc_ap( );
439     }
```

将这段代码与前面smp_boot_cpus( )开头处的代码作一比较，就可以看出这里进行的一些操作与主CPU在启动次CPU之前的操作相似，实际上也是CPU初始化的一部分。这里第421行调用calibrate_delay( )测试CPU的运算速度，主CPU则在初始化的第二阶段在start_kernel( )中调用这个函数（见第10章）。这个函数测算出CPU的“BogoMIPS”，大致上反映了以“每秒百万条指令”为单位的运算速度。有兴趣的读者不妨自己读一下。

至此，次CPU已经基本上完成了初始化，所以在432行把全局量cpu_callin_map中代表着本CPU的标志位设成1，让主CPU知道。最后，如果次CPU带有“时间印记计数器”TSC则还要通过synchronize_tsc_ap( )对其TSC进行初始化。所谓TSC是“Time Stamp Count”的缩写，这是一个64位的计数器，每来一个时钟脉冲就加1。由于是64位计数器，其数值十年以内不会重复，所以可以用它的值作为时间印记。

各个次CPU在完成了smp_callin( )的执行以后，就进入了一个无限循环，等待一个全局量smp_commenced变成1，就好像等待“起跑”命令一样，要等到了命令才起跑进入cpu_idle( )。

回到主CPU的运行中。主CPU在934～954行的for循环中启动了所有的次CPU以后，最后还有几件事要做，我们继续往下看（arch/i386/kernel/smpboot.c）。

[start_kernel( ) > smp_init( ) > smp_boot_cpus( )]

```
956         /*
957          * Cleanup possible dangling ends...
958          */
959     #ifndef CONFIG_VISWS
        . . . . . . .
974     #endif
975
976         /*
977          * Allow the user to impress friends.
978          */
979
980         Dprintk("Before bogomips.\n");
981         if (!cpucount) {
982             printk(KERN_ERR "Error: only one processor found.\n");
983         } else {
984             unsigned long bogosum = 0;
985             for (cpu = 0; cpu < NR_CPUS; cpu++)
986                 if (cpu_online_map & (1<<cpu))
987                     bogosum += cpu_data[cpu].loops_per_jiffy;
```

```
988             printk(KERN_INFO "Total of %d processors activated (%lu.%02lu BogoMIPS).\n",
989                 cpucount+1,
990                 bogosum/(500000/HZ),
991                 (bogosum/(5000/HZ))%100);
992             Dprintk("Before bogocount   setting activated=1.\n");
993         }
994         smp_num_cpus = cpucount + 1;
995 
996         if (smp_b_stepping)
997             printk(KERN_WARNING
        "WARNING: SMP operation may be unreliable with B stepping processors.\n");
998         Dprintk("Boot done.\n");
999 
1000    #ifndef CONFIG_VISWS
1001        /*
1002         * Here we can be sure that there is an IO-APIC in the system. Let's
1003         * go and set it up:
1004         */
1005        if (!skip_ioapic_setup)
1006            setup_IO_APIC( );
1007    #endif
1008 
1009        /*
1010         * Set up all local APIC timers in the system:
1011         */
1012        setup_APIC_clocks( );
1013 
1014        /*
1015         * Synchronize the TSC with the AP
1016         */
1017        if (cpu_has_tsc && cpucount)
1018            synchronize_tsc_bp( );
1019 
1020    smp_done:
1021        zap_low_mappings( );
1022    }
```

首先是显示整个系统的运算能力，即各个 CPU 运算能力的总和。然后通过 setup_IO_APIC( )对外部 APIC 进行初始化，这个函数的代码在 arch\i386\kernel\io_apic.c 中，我们只能把它留给读者。接着是对各内部 APIC 中定时器，即时钟中断源的设置，读者已经在前面看过这个函数的代码。

最后，通过 zap_low_mappings( )把页面映射目录中低区，即 3GB 以下的目录项都清除掉。这些目录项是为 CPU 由段式映射向页式映射过渡而设置的，详见第 10 章对系统初始化第一阶段的叙述。对于单 CPU 的系统，一旦 CPU 转入了页式映射，这些目录项就可以清除了。可是在 SMP 结构的系统中，则要到所有的 CPU 都转入页式映射以后才能清除。而现在是时候了。清除了低区映射以后，swapper_pg_dir 中就只剩下系统空间的映射，所以这个页面映射目录将用于所有的内核线程。同时，这个目录也是所有进程的映射目录的基础，进程的映射目录的系统空间部分就是从 swapper_pg_dir 复制

而来。

当主 CPU 从 smp_boot_cpus( )返回到 smp_init( )中时，所有的次 CPU 都已启动而先后到达同步点，已在等待最后的起跑命令了。于是，主 CPU 通过 smp_commence( )把全局量 smp_commenced 设置成 1，向所有的次 CPU 发出的起跑命令。这个函数的代码在 arch/i386/kernel/smpboot.c 中：

[start_kernel( ) > smp_init( ) > smp_commence( )]

```
164     /*
165      * Architecture specific routine called by the kernel just before init is
166      * fired off. This allows the BP to have everything in order [we hope].
167      * At the end of this all the APs will hit the system scheduling and off
168      * we go. Each AP will load the system gdt's and jump through the kernel
169      * init into idle( ). At this point the scheduler will one day take over
170      * and give them jobs to do. smp_callin is a standard routine
171      * we use to track CPUs as they power up.
172      */
173
174     static atomic_t smp_commenced = ATOMIC_INIT(0);
175
176     void __init smp_commence(void)
177     {
178         /*
179          * Lets the callins below out of their loop.
180          */
181         Dprintk("Setting commenced=1, go go go\n");
182
183         wmb( );
184         atomic_set(&smp_commenced, 1);
185     }
```

这里的 wmb( )是个内存写操作路障。将 smp_commenced 设成 1 以后，在 start_secondary( )中待命的各个次 CPU 就随之结束了 457 行的 while 循环而进入 cpu_idle( )。表面上这是个函数调用，但是实际上永远不会从那里返回了，因为 cpu_idle( )的主体是个无限循环，只要系统中有就绪进程在等待执行，就调度其运行，否则就使 CPU 进入硬件睡眠状态，直至有中断发生时才恢复运行。到那时候，既然发生了中断，就又有机会调度了。我们在下一章中还要回到这个函数代码中。

第10章

系统引导和初始化

10.1 系统引导过程概述

现代计算机系统的内存都是“挥发性”的：一旦关机断电，存储在内存中的信息、连同操作系统本身的映象就丢失了。所以，必须把操作系统(内核)的映象存储在某种不挥发的介质中，使得开机加电时有一个从不挥发介质装入操作系统、并转入运行的映象过程，这个过程就称为“引导”(bootstrap，或 boot)，也称为“自举”。这里，所谓不挥发介质通常是指硬磁盘或软盘，但也可以是 EPROM 或 Flash 存储器，还可以是网络中别的节点。一般，从 EPROM 或 Flash 存储器装入映象是很简单的，因为这些存储器本来就是内存的一部分，访问这些存储器与访问普通的内存空间并无不同。相比之下，从磁盘等外部设备装入操作系统映象就复杂多了。所以一般说的“引导”是指从磁盘上引导。我们在本节中将注意力集中在从硬盘引导，因为这是最为典型的；另一方面，理解了怎样从硬盘引导，也就不难理解怎样从别的介质引导了。

可想而知，要在开机时从不挥发介质装入操作系统的映象，系统就得要在一开机时就具有一定的智能，也就是一开机以后 CPU 就能执行一段程序，这段程序本身必须存储在作为系统内存一部分的 EPROM、Flash 等不挥发存储器中，而且必须知道怎样才能从不挥发介质装入操作系统的映象。实际上，各种 CPU 都设计成一加电以后就从某个特殊的地址开始执行指令，所以这些不挥发存储器就放在这个位置上。以 i386 CPU 为例，加电或“总清”(reset)以后 CPU 处于实地址模式，并且代码段寄存器 CS 的内容为 0xffff，而取指令指针(寄存器)IP 的内容则为 0；也就是说，从线性地址 0xffff0 开始取第一条指令。所以采用 i386 CPU 的系统在这个位置上必须有不挥发存储器。

那么，这段程序要有多大呢？这就要看具体的设计了。在早期的计算机中这段程序一般都很小，例如 2K 字节或者更小，甚至只有几条指令(记得 70 年代中美建交后进入中国的 NOVA 机，由此而来的国产机名为 DJS-130，操作系统为 RTOS，引导程序只有 13 条指令，当时称初始引导 13 条，亦称手拨 13 条。13 条指令执行结果是通过光电读入机把存放在穿孔纸带上的 RTOS 执行码装入内存)。这是因为早期的 EPROM 或 PROM 的容量都很小，并且其目的和功能也很单一。可是，这么短的一段程序怎么能把操作系统的映象从磁盘上读进来呢？我们不妨这样想：如果要把操作系统的映象作为一个文件读进来，那么这段程序就要能支持相应的文件系统，还要加上相应的设备驱动程序。一般而言，除十

分简单的文件系统以外，这是不现实的。再说，文件系统又有许多种，无论如何也不能使一小段程序同时支持好几种文件系统。所以，只能绕过文件系统，甚至绕过设备驱动的一般形式，而把操作系统映象的读入作为一个特例处理，在设备驱动的底层解决问题。例如，要是从一开始就知道操作系统的映象肯定在磁盘上开头 10 个连续的扇区中，这就比较简单了。如果不能肯定操作系统的映象一定是 10 个扇区，而且一定是前 10 个连续的扇区，那就得在磁盘上某个固定的位置(例如某一个扇区)提供有关操作系统映象的一些“地理”信息。实际上，光是提供这些“地理”信息还是不够的，因为不同文件系统所需要的信息不同，对这些信息的运用也不同。例如，在有些文件系统中，属于同一文件的所有扇区都连成一条链，所以只需提供文件中第一个扇区的位置就行了；而有些文件系统则要采用位图的方式，或者多层映射表的方式；还有，对映象还可能实施了压缩，因而需要在装入映象的同时实施解压缩；而不同 CPU 的指令系统也不同；等等，等等，不一而足。所以，除关于映象的“地理”信息以外，还需要在这个固定的位置上存放一段适用于具体操作系统的可执行程序作为补充。套用现代“面向对象的程序设计”的术语，就是要通过磁盘上的某个固定的位置，通常是一个扇区，提供一个“对象”，其中既包括了数据，也包括了运用这些数据的程序。当然，那时候还没有“对象”这么个概念，而是很自然地把这个扇区称为“引导扇区”，通常都是磁盘上的第一个扇区。而 EPROM 中的程序，则因为其作用在于从磁盘读入引导扇区，因而常常称为初始引导程序。初始引导程序是“中性”的，与具体的操作系统或文件格式无关，它只是从磁盘上读入引导扇区，然后把引导扇区的开头当作一段程序的起点，使 CPU 转入这段程序。引导扇区的内容则取决于具体的操作系统，也可能还进一步取决于文件格式。当然，引导扇区只是一个扇区，即 512 字节，能够容纳的信息和代码是很有限的，所以常常还得要由引导扇区的程序先装入其他若干扇区，再由这些扇区中的程序和数据协同完成整个引导过程。

有时候，也可以先通过引导扇区读入一个作为中间步骤的工具性程序(而不是操作系统映象)，常称为“引导装入程序”，再由引导装入程序装入操作系统映象。例如 LILO 就是 Linux 的引导装入程序。从概念上说引导装入程序与 EPROM 中的初始引导程序并无不同，只是体积更大，功能更为复杂。例如 LILO 就可以让用户从磁盘上的多个(并且可以是多种)操作系统映象中有选择地引导。另一方面，由于引导装入程序的功能较强，也有利于减小对操作系统映象在磁盘上存储位置和方式的限制。实际上，除了从软盘引导以外，Linux 一般都是通过 LILO 引导的。

随着技术的发展，像 EPROM 一类存储器件的容量愈来愈大，为加强系统初始引导程序提供了条件，所以逐渐往里面加入了如“加电自检”、“系统配置”一类的功能，(初始的)人机交互界面也逐渐得到改善。到 Microsoft 为 IBM PC 机设计 DOS 操作系统(DOS 是“磁盘操作系统”的缩写)时，更是把 DOS 的整个设备驱动层都放在了 EPROM 中，称为“基本输入/输出系统”，即 BIOS。以后，这种格局就一直沿用了下来。在 BIOS 中，初始引导只是其功能的一部分，而且只是很小的一部分。现在的 BIOS 甚至已经比当初的整个 DOS 还要大了，可是再大也不可能把所有操作系统的引导都考虑进去，所以还是采用初始引导程序加引导扇区的方案，让各种操作系统通过引导扇区进一步提供其自身的引导手段。其实，虽然 BIOS 提供了对各种主要外部设备的基本驱动，但是，对于多进程、多用户、采用保护模式尤其是页式映射的操作系统却未必合适，所以 Linux 内核并不使用 BIOS 所提供的设备驱动，而是绕开 BIOS，从硬件接口(寄存器)和中断响应开始彻底地实现自己的设备驱动层。对于 Linux 来说，BIOS 的作用只不过就是初始引导。如果要说其他还有什么作用的话，那就是加电以后的自检以及向内核提供在此过程中搜集到的一些信息了(有些信息是由用户设置的)。

另一方面，由于硬盘容量的迅速发展，实际使用中常常把一个硬盘划分成若干“分区”，从而把一个物理的硬盘划分成若干个逻辑磁盘。这样一来，每个逻辑磁盘的第一个扇区仍然是引导扇区，分别

用于相应逻辑磁盘中的操作系统映象(如果有的话)。但是，这些引导扇区在物理上当然不再是整个硬盘的第一个扇区了。在这种情况下，整个硬盘的第一个扇区不属于任何一个逻辑磁盘，或者说已经超脱于所有这些逻辑磁盘之外，上了一个层次。但是，BIOS 还是把它当作整个硬盘的引导扇区，加电时还是从这个扇区“引导”，所以这个扇区称为“主引导记录块(扇区)”MBR。MBR 中含有关于盘区划分的信息(通过 fdisk 设置)，还有一段简短的程序，一共 512 字节。不过，MBR 中的程序并不直接引导操作系统，而是根据盘区划分的信息从一个预定的“活跃”逻辑磁盘中读入其引导扇区，再由这个引导扇区自己采取进一步的行动。所以，可以把 MBR 的作用看成是为 BIOS 中的初始引导程序提供了一种类似于间接寻址的手段。不过，也可以把用来“引导”LILO 的程序(连同有关盘区划分的信息)放在 MBR 中，使整个引导过程少转一道弯。

可想而知，引导扇区中的程序及其辅助程序(不包括 LILO)必须是很精练的，所以都采用汇编语言编写。这些源代码都在 arch/ 下面具体 CPU 名下的 boot 目录中。就 i386 而言，都在 arch/i386/boot 中。这个目录中有三个汇编语言的程序：

(1) bootsect.S，这是 Linux 引导扇区的源代码，汇编以后不得超过 512 字节。

(2) setup.S，这是辅助程序的一部分。

(3) video.S，这是辅助程序的另一部分，用于引导过程中的屏幕显示。

在 arch/i386/boot 中还有个子目录 compressed，含有两个源代码文件 head.S 以及 misc.c。版本较新的内核映象都是经过压缩的，在引导的过程中要解压缩，这两个文件(还有 lib/inflate.c)中的代码就是用于解压缩，也属于辅助程序的一部分。这样，经过编译、汇编、连接以后就形成三个组成部分，即引导扇区的映象 bootsect、辅助程序 setup 以及内核映象本身(通常是 vmlinux)。

严格说来，bootsect 和 setup 并不是内核的一部分，它们的代码又都是用汇编语言编写的，篇幅很大，所以我们在这里只作一些简短的说明，而不深入到这些代码中去了，有兴趣或需要的读者可以自己阅读。这些代码都有比较好的注释，读起来不会太困难。

引导扇区代码 arch/i386/boot/bootsect.S 的作者在文件开头处的注释中，对开始执行这段代码以后的过程作了说明：

```
1     /*
2      *  bootsect.S      Copyright (C) 1991, 1992 Linus Torvalds
3      *
4      *  modified by Drew Eckhardt
5      *  modified by Bruce Evans (bde)
6      *  modified by Chris Noe (May 1999) (as86 -> gas)
7      *
8      * bootsect is loaded at 0x7c00 by the bios-startup routines, and moves
9      * itself out of the way to address 0x90000, and jumps there.
10     *
11     * bde - should not jump blindly, there may be systems with only 512K low
12     * memory. Use int 0x12 to get the top of memory, etc.
13     *
14     * It then loads 'setup' directly after itself (0x90200), and the system
15     * at 0x10000, using BIOS interrupts.
16     *
17     * NOTE! currently system is at most (8*65536-4096) bytes long. This should
18     * be no problem, even in the future. I want to keep it simple. This 508 kB
```

```
19      * kernel size should be enough, especially as this doesn't contain the
20      * buffer cache as in minix (and especially now that the kernel is
21      * compressed :-)
22      *
23      * The loader has been made as simple as possible, and continuous
24      * read errors will result in a unbreakable loop. Reboot by hand. It
25      * loads pretty fast by getting whole tracks at a time whenever possible.
26      */
```

读者大概还是不甚了了，我们再作一些解释。

在 PC 的系统结构中，线性地址 0xA0000 以上，即 640KB 以上都用于图形接口卡以及 BIOS 本身，而 0xA0000 以下的 640KB 为系统的基本内存。如果配备更多的内存，则从 0x100000，即 1MB 处开始，称为“高内存”。当 BIOS(或 LILO)“引导”一个系统时，总是把引导扇区读入到基本内存中地址为 0x7c00 的地方，然后就跳转到 0x7c00 开始执行引导扇区的代码。这段代码将其自身“搬运”到 0x90000 处，并且跳转到那里继续执行，然后通过 BIOS 提供的读磁盘调用“int 0x13”从磁盘上读入 setup 和内核的映象。其中 setup 的映象读入到地址为 0x90200 的地方，就是经过“搬运”后 bootsect 所在处的上方。然后，就跳转到 setup 的代码中，作好执行内核映象的准备，包括映象的解压缩(如果需要的话)，从 BIOS 收集一些数据，在控制台上显示一些信息等等。从 0x90000 到 0xA0000 一共是 64KB，bootsect 只占 512 字节，所以 setup 的大小理论上可达 63.5KB（实际上当然不会那么大）。

基本内存中开头一部分空间是保留给 BIOS 自己用的，另一方面对于 Linux 内核的引导也需要保留一些运行空间，所以一共保留了 64KB。这样，基本内存中剩下来可用于内核映象的就是 8 个 64KB，即 512KB。但是，在这 512KB 的顶端还要留下 4KB 用于引导命令行(LILO 支持引导命令行)以及一些需要传递给内核的数据(从 BIOS 收集得到)。于是，基本内存中实际可用于内核映象的空间就是 508KB。内核映象一般都是经过压缩的，压缩以后的内核映象就像是一大块数据，跟引导扇区和引导辅助程序的映象拼接在一起，成为内核的“引导映象”。目录 arch/i386/boot/tools 中的 build.c 经编译/连接以后产生的可执行程序 build，就是用来拼接引导映象的工具。大小不超过 508KB 的引导映象称为“小映象”，文件名为 zImage；否则就称为“大内核”，文件名为 bzImage。由于 bzImage 在基本内存中已经装载不下，所以要装载在地址为 0x100000（1MB）的地方。不过，不管是 zImage 还是 bzImage，解除压缩以后的内核映象总是放在地址为 0x100000（1MB）的地方。

CPU 在跳转到 bootsect 时尚处于 16 位实地址模式，然后在 setup 的执行过程中转入 32 位保护模式的段式寻址方式。在 bootsect 和 setup 的执行中，二者都利用 BIOS 提供的调用来完成一些比较大的操作，如读磁盘，取得 BIOS 在加电自检时搜集到的有关内存的信息等等。一旦转入内核映象本身的执行，就与 BIOS 分道扬镳，不再使用 BIOS 调用了。

辅助程序 setup 为内核映象的执行作好了准备(包括解除压缩)以后，就跳转到 0x100000 开始内核本身的执行，此后就是内核的初始化过程了。内核的初始化是个非常漫长的过程，整个过程可以分成三个阶段。第一个阶段主要是 CPU 本身的初始化，例如页式映射的建立；第二个阶段主要是系统中一些基础设施的初始化，例如内存管理和进程管理的建立和初始化；最后则是“上层建筑”的初始化，如根设备的安装和外部设备的初始化等等。由于整个过程涉及的代码太大，我们相应地把它分成三节加以叙述。

10.2　系统初始化（第一阶段）

将 Linux 内核的映象装入内存中，并且作好了一些必要的准备以后，CPU 就通过一条长程转移指令转到映象代码段开头的入口 startup_32，从那里开始执行。对于 SMP 结构的系统，这个时候在运行的只是其中的一个处理器，就是所谓“主 CPU”（也称“引导处理器”BP），而其他的“次 CPU”（也称“应用处理器”AP）则处于停机状态，等待主 CPU 的启动。次 CPU 在受到启动进入内核时同样也要从 startup_32 开始执行，所以从 startup_32 开始的代码是公共的。但是有些操作仅由主 CPU 执行，另有一些操作则仅由次 CPU 执行。主 CPU 在进入 startup_32 时其寄存器%bx 的内容为 0，而次 CPU 在进入 startup_32 时其寄存器%bx 的内容为 1，在执行的过程中就据此区分执行者(CPU)在系统中扮演的角色。不过，这里要着重指出，这里的代码虽然是公共的，但是并不意味着主 CPU 和次 CPU 有可能并发地执行这段程序。实际上，主 CPU 是开路先锋，它首先执行这段程序，“逢山开路，遇水搭桥”，完成以后才逐个地启动次 CPU 执行，并且等待其完成。所以，在同一时间中，系统中最多只有一个处理器在执行这段程序。

内核映象的起点是 stext，也是_stext，引导和解压缩以后的整个映象放在内存中从 0x100000 即 1MB 开始的区间。CPU 执行内核映象的入口 startup_32 就在内核映象开头的地方，因此其物理地址也是 0x100000。然而，读者在第 2 章中曾看到，在正常运行时整个内核映象都应该在系统空间中，系统空间的地址映射是线性的、连续的，虚拟地址与物理地址间有个固定的位移，这就是 0xC0000000，即 3GB。所以，在连接内核映象时已经在所有的符号地址上加了一个偏移量 0xC0000000，这样 startup_32 的虚拟地址就成了 0xC0100000。

不管是主 CPU 还是次 CPU，进入 startup_32 时都运行于保护模式下的段式寻址方式。段描述表中与__KERNEL_CS 和__KERNEL_DS 相对应的描述项所提供的基地址都是 0，所以实际产生的就是第 2 章中讲过的“线性地址”。其中代码段寄存器 CS 已在进入 startup_32 之前设置成__KERNEL_CS，数据段寄存器则尚未设置成__KERNEL_DS。不过，虽然代码段寄存器已经设置成__KERNEL_CS，从而 startup_32 的地址为 0xC0100000。但是在转入这个入口时使用的指令是“ljmp 0x100000”而不是“ljmp startup_32”，所以装入 CPU 中寄存器 IP 的地址是物理地址 0x100000 而不是虚拟地址 0xC0100000。这样，CPU 在进入 startup_32 以后就会继续以物理地址取指令。只要不在代码段中引用某个地址，例如向某个地址作绝对转移，或者调用某个子程序，就可以一直这样运行下去，而与 CS 的内容无关。此外，CPU 的中断已在进入 startup_32 之前关闭。

从 startup_32 开始的汇编代码在 arch/i386/kernel/head.S 中，这就是初始化的第一阶段。

[startup_32( )]

```
39    /*
40     * swapper_pg_dir is the main page directory, address 0x00101000
41     *
42     * On entry, %esi points to the real-mode code as a 32-bit pointer.
43     */
44    ENTRY(stext)
45    ENTRY(_stext)
46    startup_32:
```

```
47    /*
48     * Set segments to known values
49     */
50        cld
51        movl $(__KERNEL DS),%eax
52        movl %eax,%ds
53        movl %eax,%es
54        movl %eax,%fs
55        movl %eax,%gs
56    #ifdef CONFIG_SMP
57        orw %bx,%bx
58        jz 1f
59
60    /*
61     *  New page tables may be in 4Mbyte page mode and may
62     *  be using the global pages.
63     *
64     *  NOTE! If we are on a 486 we may have no cr4 at all!
65     *  So we do not try to touch it unless we really have
66     *  some bits in it to set.  This won't work if the BSP
67     *  implements cr4 but this AP does not -- very unlikely
68     *  but be warned!  The same applies to the pse feature
69     *  if not equally supported. --macro
70     *
71     *  NOTE! We have to correct for the fact that we're
72     *  not yet offset PAGE_OFFSET..
73     */
74    #define cr4_bits  mmu_cr4_features-__PAGE_OFFSET
75        cmpl $0,cr4_bits
76        je 3f
77        movl %cr4,%eax        # Turn on paging options (PSE,PAE,..)
78        orl cr4_bits,%eax
79        movl %eax,%cr4
80        jmp 3f
81    1:
82    #endif
83    /*
84     * Initialize page tables
85     */
86        movl $pg0-__PAGE OFFSET,%edi /* initialize page tables */
87        movl $007,%eax        /* "007" doesn't mean with right to kill, but
88                        PRESENT+RW+USER */
89    2:  stosl
90        add $0x1000,%eax
91        cmp $empty_zero_page-__PAGE_OFFSET,%edi
92        jne 2b
93
```

首先将 CPU 中除 CS 以外所有的段寄存器，即%ds、%es、%fs 以及%gs，都设置成__KERNEL_DS，读者在第 2 章中看到过这个数值是 0x18，表示采用全局段描述表 GDT 中下标为 3 的段描述项，而 RPL 则为 0。引导辅助程序在转入 startup_32 之前准备下了一个临时的全局段描述表，表中相应的描述项给出基地址为 0，也就是使用线性地址。

不管是主 CPU 还是次 CPU，在 Linux 内核中都要进入页式映射模式运行。暂时以物理地址取指令虽然是可以的，但是不能在代码中向符号地址作绝对转移或调用子程序，因此终非长久之计，所以，要尽快准备好页面映射表并开启 CPU 的页面映射机制。代码中的 86～92 行将从 pg0 开始直到 empty_zero_page 之间的 8K 字节设置成一个临时的页面映射表。这个页面表在内存中，是由所有 CPU 公用的，所以只由主 CPU 进行初始化，次 CPU 则要跳过这一小段代码(见 57～58 行以及 76 和 80 行)。此外，对于次 CPU 这里还有个小插曲，那就是如果系统支持 PSE/PAE，即 36 位地址模式的话还要相应地设置其控制寄存器%cr4（77～79 行）。

```
399     /*
400      * The page tables are initialized to only 8MB here - the final page
401      * tables are set up later depending on memory size.
402      */
403     .org 0x2000
404     ENTRY(pg0)
405
406     .org 0x3000
407     ENTRY(pg1)
408
409     /*
410      * empty_zero_page must immediately follow the page tables ! (The
411      * initialization loop counts until empty_zero_page)
412      */
413
414     .org 0x4000
415     ENTRY(empty_zero_page)
```

常数__PAGE_OFFSET 为系统空间的虚拟地址与物理地址之间的差距，定义于 include/asm-i386/page.h：

```
 81     #define __PAGE_OFFSET         (0xC0000000)
```

可见，pg0 在地址(相对于程序的起点，见下面)为 0x2000 的地方。这个页面映射表中的表项依次设置为 0x7、0x1007、0x2007 等等。其中最低的三位均为 1，表示页面为用户页面，可写，并且页面的内容在内存中(参阅第 2 章)。映射的目标，即物理页面的基地址，则分别为 0x0、0x1000、0x2000 等等，也就是物理内存中的页面 0、1、2 等等。映射表的大小是两个页面，即 2K 个表项，所以代表着一块 8MB 的存储空间，这就是 Linux 内核对内存大小的最低限度要求。

那么，页面目录在哪儿呢？

```
383     /*
384      * This is initialized to create an identity-mapping at 0-8M (for bootup
```

```
385      * purposes) and another mapping of the 0-8M area at virtual address
386      * PAGE_OFFSET.
387      */
388     .org 0x1000
389     ENTRY(swapper_pg_dir)
390         .long 0x00102007
391         .long 0x00103007
392         .fill BOOT_USER_PGD_PTRS-2,4,0
393         /* default: 766 entries */
394         .long 0x00102007
395         .long 0x00103007
396         /* default: 254 entries */
397         .fill BOOT_KERNEL_PGD_PTRS-2,4,0
```

这里的 392 行利用汇编语言提供的功能在其所在的位置上填入 766 个目录项，每个目录项的大小为 4 个字节，内容为 0。同样，397 行也填入 254 个这样的目录项。这里引用的几个常数均定义于 include/asm-i386/pgtable.h：

```
126     #define TWOLEVEL_PGDIR_SHIFT      22
127     #define BOOT_USER_PGD_PTRS        (__PAGE_OFFSET >> TWOLEVEL_PGDIR_SHIFT)
128     #define BOOT_KERNEL_PGD_PTRS      (1024-BOOT_USER_PGD_PTRS)
```

回顾一下第 2 章的内容，这里的常数 __PAGE_OFFSET 是 0xC0000000，所以 BOOT_USER_PGD_PTRS 为 768，而 BOOT_KERNEL_PGD_PTRS 则为 256。一个页面目录的大小是 4KB，含有 1024 个目录项，共代表着 4GB 的虚存空间。Linux 内核以 3GB 为界把整个虚存空间分成用户空间和系统空间两部分。所以，页面目录中的低 768 个目录项用于用户空间的映射，而高 256 个目录项用于系统空间的映射。在初始的页面目录 swapper_pg_dir 中，用户空间和系统空间都只映射了开头的两个目录项（390~391 行以及 394~395 行），即 8MB 的空间，而且有着相同的映射，即指向相同的页面映射表。这样，以符号地址 empty_zero_page 为例，在连接内核映象时这个地址显然是安排在 0xC0000000 以上的空间，而物理上却是装入在(empty_zero_page－0xC0000000)的地址上。所以，若在未开启页式映射之前要引用这个符号，就要从中减去位移量__PAGE_OFFSET（见 86 行和 91 行）。可是，一旦开启页式映射以后，再要引用这个符号时就可以(而且应该)直接引用了。再来看目录项的内容，以 394 行为例，这个目录项指向物理地址 0x00102000，这是在 1MB 以上的区间中，实际上就是 pg0 的物理地址。前面 403 和 404 行指定 pg0 的起点地址为 0x2000，那是假定整个内核映象是从地址 0 开始时的起点，而现在内核映象放在从地址 0x00100000 开始的地方，pg0 的起点也就相应地变成了 0x00102000。

那么，为什么在虚存空间的低区，即本来应该属于用户空间的位置上(390~391 行)也要放上同样的目录项呢？简而言之是为了平稳过渡。如上所述，CPU 进入 startup_32( )以后是以物理地址来取指令的。在这种情况下，如果映射目录只包括系统空间，即虚存空间高区的映射，而不包括虚存空间低区的映射，则一旦开启页式映射以后就不能继续执行了，因为此时 CPU 中的取指令指针 IP 仍指向低区，仍会以物理地址取指令，直到以某个符号地址为目标作绝对转移或调用子程序时为止。所以，解决的办法是分两步走：

(1) 先开启页式映射，但是在虚存空间的低区暂时提供与高区相同的映射，使 CPU 可以继续执行。

(2) 在开启了页式映射之后，以一个符号地址为目标执行一条绝对转移指令。如前所述，由符号所代表的地址是系统空间的虚拟地址。CPU 在执行绝对转移指令时把这个虚拟地址装入 IP，从此就改成以虚拟地址在系统空间取指令了。

CPU 转入系统空间以后，应该把低区的映射清除。后面读者将会看到，页面映射目录 swapper_pg_dir 经过扩充以后就成为所有内核线程的页面映射目录。在内核线程的正常运行中，处于系统状态的 CPU 是不应该通过用户空间的虚拟地址访问内存的。清除了低区的映射以后，如果发生 CPU 在内核中通过用户空间的虚拟地址访问内存，就可以因为产生页面异常而抓住这个错误。

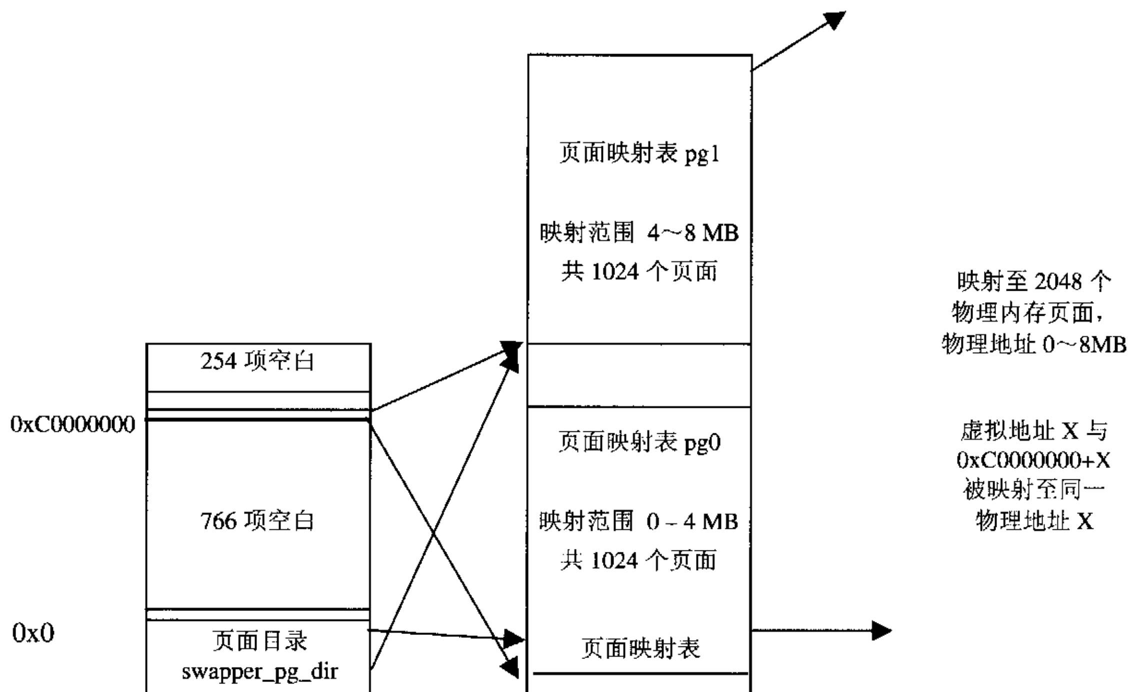


图 10.1　初始化第一阶段的页面映射

在开启页式映射完成向系统空间过渡之前，如果必须要作绝对转移或系统调用，就得在目标地址上减去位移量__PAGE_OFFSET，即 0xC0000000。就凭这一点，向页式映射和系统空间的过渡就宜早不宜迟。所以，现在是开启页面映射机制的时候了。注意下面又是主 CPU 和次 CPU 共用的代码了。页面映射是 CPU 内部的功能，每个 CPU 都要开启其自己的页面映射机制，尽管使用的页面映射目录和页面映射表是相同的。

[startup_32( )]

```
 94    /*
 95     * Enable paging
 96     */
 97    3:
 98        movl $swapper_pg_dir-__PAGE_OFFSET,%eax
 99        movl %eax,%cr3       /* set the page table pointer.. */
100        movl %cr0,%eax
```

```
101         orl $0x80000000,%eax
102         movl %eax,%cr0       /* ..and set paging (PG) bit */
103         jmp 1f           /* flush the prefetch-queue */
104     1:
105         movl $1f,%eax
106         jmp *%eax        /* make sure eip is relocated */
107     1:
108         /* Set up the stack pointer */
109         lss stack_start,%esp
110
```

将页面目录的地址装入控制寄存器%cr3，并把%cr0 中的最高位设置成 1，就开启了 CPU 的页面映射机制。装入控制寄存器%cr3 的地址必须是页面目录的物理地址，所以要从虚拟地址 swapper_pg_dir 中减去偏移量，把它还原为物理地址。

开启页面映射机制以后立即就有条相对转移指令(103～104 行)，从逻辑上说这条指令不起什么作用，但是它起到了丢弃已经在 CPU 的取指令流水线中内容的作用，这是 Intel 在 i386 的技术资料中建议的。由于跳转的目标离得很近，103 行的 jmp 指令是条相对转移指令，只是在 IP 的当前值上加了一个不大的位移，CPU 仍旧以物理地址取指令而并没有转入系统空间，所以至 104 行为止只是完成了上述的第一步。但是，从此开始，通过数据段引用符号地址时就不需要从中减去偏移量__PAGE_OFFSET 了。紧接着的另一条转移指令(106 行)就不同了，105 行把目标地址置入%eax 时是通过数据段引用这个符号的，所以这个地址在系统空间中，然后以%eax 中的这个地址为目标执行 jmp 指令，就使 CPU 转入了系统空间，完成了两种模式间的平稳过渡。

本来，现在可以清除页面映射目录低区的那些目录项了，可是，考虑到次 CPU 将来也要有这么个过渡的过程，所以还得暂时保留着，到系统中所有的 CPU 都完成了过渡才来清除。

代码中的 109 行将 CPU 的堆栈设置在 stack_start 处。

```
330     ENTRY(stack_start)
331         .long SYMBOL_NAME(init_task_union)+8192
332         .long __KERNEL_DS
```

这里的 init_task_union 是个 union，其类型定义于 include/linux/sched.h：

```
480     #ifndef INIT_TASK_SIZE
481     # define INIT_TASK_SIZE 2048*sizeof(long)
482     #endif
483
484     union task_union {
485         struct task_struct task;
486         unsigned long stack[INIT_TASK_SIZE/sizeof(long)];
487     };
```

也就是说，既可以把 task_union 看成一个 task_struct 结构，也可以把它看成一个数组，数组的大小是 8K 字节，即两个页面。读者在第 4 章中看到，每个进程的 task_struct 数据结构和系统空间堆栈共占两个页面，下部是 task_struct 结构，上部是系统空间堆栈。可想而知，这里是在为创建系统中的第一

个进程进行准备，而具体的数据结构 init_task_union 则定义于 arch/i386/kernel/init_task.c：

```
15    /*
16     * Initial task structure.
17     *
18     * We need to make sure that this is 8192-byte aligned due to the
19     * way process stacks are handled. This is done by having a special
20     * "init_task" linker map entry..
21     */
22    union task_union init_task_union
23        __attribute__((__section__(".data.init_task"))) =
24            { INIT_TASK(init_task_union.task) };
```

数据结构的内容由宏定义 INIT_TASK( )决定，其代码在 include/linux/sched.h 中：

```
434   /*
435    *  INIT_TASK is used to set up the first task table, touch at
436    * your own risk!. Base=0, limit=0x1fffff (=2MB)
437    */
438   #define INIT_TASK(tsk)  \
439   {                              \
440       state:       0,                    \
441       flags:       0,                    \
442       sigpending: 0,                     \
443       addr_limit: KERNEL_DS,                   \
444       exec_domain:    &default_exec_domain,                  \
445       lock_depth: -1,                    \
446       counter:         DEF_COUNTER,                   \
447       nice:        DEF_NICE,                   \
448       policy:      SCHED_OTHER,                  \
449       mm:          NULL,                      \
450       active_mm:       &init_mm,                  \
451       cpus_allowed:        -1,                   \
452       run_list:        LIST_HEAD_INIT(tsk.run_list),           \
453       next_task:       &tsk,                  \
454       prev_task:       &tsk,                  \
455       p_opptr:         &tsk,                  \
456       p_pptr:      &tsk,                   \
457       thread_group:        LIST_HEAD_INIT(tsk.thread_group),        \
458       wait_chldexit:\ __WAIT_QUEUE_HEAD_INITIALIZER(tsk.wait_chldexit),\
459       real_timer:      {                       \
460           function:        it_real_fn            \
461       },                                 \
462       cap_effective:  CAP_INIT_EFF_SET,                 \
463       cap_inheritable:     CAP_INIT_INH_SET,                 \
464       cap_permitted:  CAP_FULL_SET,                 \
465       keep_capabilities:       0,                \
```

```
466         rlim:         INIT_RLIMITS,                      \
467         user:         INIT_USER,                     \
468         comm:         "swapper",                     \
469         thread:       INIT_THREAD,                       \
470         fs:           &init_fs,                      \
471         files:        &init_files,                       \
472         sigmask_lock:        SPIN_LOCK_UNLOCKED,                   \
473         sig:          &init_signals,                     \
474         pending:          { NULL,  &tsk.pending.head,  {{0}}},         \
475         blocked:          {{0}},                         \
476         alloc_lock:      SPIN_LOCK_UNLOCKED                  \
477     }
```

这个进程的 thread 结构定义为全 0:

```
379     #define INIT_THREAD  {                         \
380         0,                           \
381         0,  0,  0,  0,                       \
382         { [0 ... 7] = 0 },  /* debugging registers */
383         0,  0,  0,                       \
384         { { 0, }, },          /* 387 state */          \
385         0,0,0,0,0,0,                          \
386         0, {~0,}            /* io permissions */         \
387     }
```

显然，这个进程的名称是“swapper”。代码的作者在注释中警告不要轻易改变它的内容。读者还要注意，不要把这个进程与后面讲到的 init 进程相混淆。

读者也许要问，上面 98～109 行这段代码是共同的，如果主 CPU 和次 CPU 都把系统空间堆栈设置在同一个地方，难道就不会引起冲突吗？不会，这两个页面的使用只是暂时的，在同一时间中只可能有一个 CPU 在使用，而且用完了就不会再回来，所以不会造成问题。读者在后面就会看到，次 CPU 在经由 initialize_secondary( )进入 start_secondary( )的途中会将其系统空间堆栈(从而其 task_struct 结构)更换到由主 CPU 为之准备好的地方。

[startup_32( )]

```
111     #ifdef CONFIG_SMP
112         orw  %bx,%bx
113         jz   1f                 /* Initial CPU cleans BSS */
114         pushl $0
115         popfl
116         jmp checkCPUtype
117     1:
118     #endif CONFIG_SMP
119
120     /*
121      * Clear BSS first so that there are no surprises...
```

```
122      * No need to cld as DF is already clear from cld above...
123      */
124         xorl %eax,%eax
125         movl $ SYMBOL_NAME(__bss_start),%edi
126         movl $ SYMBOL_NAME(_end),%ecx
127         subl %edi,%ecx
128         rep
129         stosb
130
131     /*
132      * start system 32-bit setup. We need to re-do some of the things done
133      * in 16-bit mode for the "real" operations.
134      */
135       call setup_idt
```

对于次 CPU，只是在这里的114～116行将其“标志寄存器”设置成0，接着就转到下面的标号 checkCPUtype 处了。而主 CPU 则担任着开路先锋的角色，需要作更大的贡献。

第一件事是初始化内核的 bss 段。内核的映象也跟其他的可执行程序一样有个 bss 段，bss 段中是一些全局变量或者静态(static)变量，需要在开始运行程序的主体之前将这个区间全部清0。这里的124～129行把从__bss_start 开始到_end 为止的 bss 段全部清0。顺便提一下，像__bss_start、_end 这些符号的值是由 gcc 在编译和连接时自动生成的。

第二件事是通过 setup_idt( )设置初始状态的中断向量表，或曰中断描述表。读者在第3章中已经看到，每个表项的大小是8个字节，共有256个表项。函数 setup_idt( )的代码在同一个文件中。

[startup_32( ) > setup_idt( )]

```
304     /*
305      *  setup_idt
306      *
307      *  sets up a idt with 256 entries pointing to
308      *  ignore_int, interrupt gates. It doesn't actually load
309      *  idt - that can be done only after paging has been enabled
310      *  and the kernel moved to PAGE_OFFSET. Interrupts
311      *  are enabled elsewhere, when we can be relatively
312      *  sure everything is ok.
313      */
314     setup_idt:
315         lea ignore_int,%edx
316         movl $(__KERNEL_CS << 16),%eax
317         movw %dx,%ax          /* selector = 0x0010 = cs */
318         movw $0x8E00,%dx      /* interrupt gate - dpl=0, present */
319
320         lea SYMBOL_NAME(idt_table),%edi
321         mov $256,%ecx
322     rp_sidt:
323         movl %eax,(%edi)
```

```
324         movl %edx,4(%edi)
325         addl $8,%edi
326         dec %ecx
327         jne rp_sidt
328         ret
```

初始状态的 256 个中断描述项全都相同，都指向同一个中断响应程序 ignore_int( )。代码的 315～318 行在寄存器 %edx 和 %eax 中构筑起一个中断门的映象，然后就通过一个循环将相同的内容写入从 idt_table 开始的 256 个表项中。读者不妨结合第 3 章中的有关内容搞清楚：这些表项的函数指针指向 ignore_int( )，而 P 标志位为 1(表示在内存中)，DPL 为 0(级别最高)，D 标志位为 1(32 位)，类型码为 110(中断门)，合在一起就是 0x8e00。此外，读者在第 3 章中已经看到，这些表项的内容以后可以通过 set_trap_gate( )、set_system_gate( )等函数加以改变。中断响应程序 ignore_int( )的代码也在同一文件中。

```
337         ALIGN
338     ignore_int:
339         cld
340         pushl %eax
341         pushl %ecx
342         pushl %edx
343         pushl %es
344         pushl %ds
345         movl $(__KERNEL_DS),%eax
346         movl %eax,%ds
347         movl %eax,%es
348         pushl $int_msg
349         call SYMBOL_NAME(printk)
350         popl %eax
351         popl %ds
352         popl %es
353         popl %edx
354         popl %ecx
355         popl %eax
356         iret
```

```
334     /* This is the default interrupt "handler" :-) */
335     int_msg:
336         .asciz "Unknown interrupt\n"
```

就是说，如果发生中断就通过 printk( )在屏幕上显示一行出错信息。在初始化期间，printk( )在屏幕上显示信息，而在系统转入正式运行以后则一般都是将信息写入系统的运行日志/var/messages。至于中断描述表 idt_table[ ]，则是在 arch/i386/kernel/traps.c 中定义的全局量。

```
58      /*
59       * The IDT has to be page-aligned to simplify the Pentium
60       * F0 0F bug workaround.. We have a special link segment
61       * for this.
```

```
62      */
63      struct desc_struct idt_table[256]
                  __attribute__ ((__section__(".data.idt"))) = { {0, 0}, };
```

不过，这里还只是为这种中断响应作了一些准备，尚未把中断描述表的地址设置进“中断描述表寄存器”IDTR，也没有打开中断。

第三件事与引导命令行和参数有关。如前所述，LILO 允许在引导时使用命令行，并将其传递给内核，而 setup 则从 BIOS 收集一些数据，作为“引导参数”一起传递给引导进来的内核。这些数据合在一起占据不超过一个页面的空间，只是在初始化期间才用到。内核中本来有个内容为全 0 的页面 empty_zero_page，代码中常常通过宏定义 ZERO_PAGE 引用的就是这个页面。不过，这个页面要到初始化完成、系统转入正常运行时才会用到，现在不妨先利用一下，所以把命令行和引导参数都复制到这个页面中。这样，这些数据原来占据的页面就腾了出来，可以回收了(而 empty_zero_page 反正是不能回收的)。

[startup_32( )]

```
136     /*
137      * Initialize eflags.  Some BIOS's leave bits like NT set.  This would
138      * confuse the debugger if this code is traced.
139      * XXX - best to initialize before switching to protected mode.
140      */
141         pushl $0
142         popfl
143     /*
144      * Copy bootup parameters out of the way. First 2kB of
145      * _empty_zero_page is for boot parameters, second 2kB
146      * is for the command line.
147      *
148      * Note: %esi still has the pointer to the real-mode data.
149      */
150         movl $ SYMBOL_NAME(empty_zero_page),%edi
151         movl $512,%ecx
152         cld
153         rep
154         movsl
155         xorl %eax,%eax
156         movl $512,%ecx
157         rep
158         stosl
159         movl SYMBOL_NAME(empty_zero_page)+NEW_CL_POINTER,%esi
160         andl %esi,%esi
161         jnz 2f            # New command line protocol
162         cmpw $(OLD_CL_MAGIC),OLD_CL_MAGIC_ADDR
163         jne 1f
164         movzwl OLD_CL_OFFSET,%esi
165         addl $(OLD_CL_BASE_ADDR),%esi
```

```
166     2:
167         movl $ SYMBOL_NAME(empty_zero_page)+2048,%edi
168         movl $512,%ecx
169         rep
170         movsl
171     1:
```

将 setup 传递过来的引导参数和命令行复制到 empty_zero_page 中以后，就在这个页面中形成了一个“参数块”，其内容如下(见 arch/i386/kernel/setup.c)：

```
151     /*
152      * This is set up by the setup-routine at boot-time
153      */
154     #define PARAM   ((unsigned char *)empty_zero_page)
155     #define SCREEN_INFO (*(struct screen_info *) (PARAM+0))
156     #define EXT_MEM_K (*(unsigned short *) (PARAM+2))
157     #define ALT_MEM_K (*(unsigned long *) (PARAM+0x1e0))
158     #define E820_MAP_NR (*(char*) (PARAM+E820NR))
159     #define E820_MAP    ((struct e820entry *) (PARAM+E820MAP))
160     #define APM_BIOS_INFO (*(struct apm_bios_info *) (PARAM+0x40))
161     #define DRIVE_INFO (*(struct drive_info_struct *) (PARAM+0x80))
162     #define SYS_DESC_TABLE (*(struct sys_desc_table_struct*)(PARAM+0xa0))
163     #define MOUNT_ROOT_RDONLY (*(unsigned short *) (PARAM+0x1F2))
164     #define RAMDISK_FLAGS (*(unsigned short *) (PARAM+0x1F8))
165     #define ORIG_ROOT_DEV (*(unsigned short *) (PARAM+0x1FC))
166     #define AUX_DEVICE_INFO (*(unsigned char *) (PARAM+0x1FF))
167     #define LOADER_TYPE (*(unsigned char *) (PARAM+0x210))
168     #define KERNEL_START (*(unsigned long *) (PARAM+0x214))
169     #define INITRD_START (*(unsigned long *) (PARAM+0x218))
170     #define INITRD_SIZE (*(unsigned long *) (PARAM+0x21c))
171     #define COMMAND_LINE ((char *) (PARAM+2048))
172     #define COMMAND_LINE_SIZE 256
```

随着代码的阅读，读者自会慢慢明白这些数据的用途。

至此，需要由主 CPU 单独进行的操作，即上述的三件事情都已经完成了。从标号 checkCPUtype 开始，又是所有 CPU 公共的代码了(见 116 行的 jmp 指令)。我们继续往下看。

[startup_32( )]

```
172     #ifdef CONFIG_SMP
173     checkCPUtype:
174     #endif
175
176         movl $-1,X86_CPUID      #  -1 for no CPUID initially
177
178     /* check if it is 486 or 386. */
179     /*
```

```
180     * XXX - this does a lot of unnecessary setup.  Alignment checks don't
181     * apply at our cpl of 0 and the stack ought to be aligned already, and
182     * we don't need to preserve eflags.
183     */
184
185         movl $3,X86         # at least 386
186         pushfl              # push EFLAGS
187         popl %eax           # get EFLAGS
188         movl %eax,%ecx          # save original EFLAGS
189         xorl $0x40000,%eax  # flip AC bit in EFLAGS
190         pushl %eax          # copy to EFLAGS
191         popfl               # set EFLAGS
192         pushfl              # get new EFLAGS
193         popl %eax           # put it in eax
194         xorl %ecx,%eax          # change in flags
195         andl $0x40000,%eax  # check if AC bit changed
196         je is386
197
198         movl $4,X86         # at least 486
199         movl %ecx,%eax
200         xorl $0x200000,%eax # check ID flag
201         pushl %eax
202         popfl               # if we are on a straight 486DX, SX, or
203         pushfl              # 487SX we can't change it
204         popl %eax
205         xorl %ecx,%eax
206         pushl %ecx          # restore original EFLAGS
207         popfl
208         andl $0x200000,%eax
209         je is486
210
211         /* get vendor info */
212         xorl %eax,%eax              # call CPUID with 0 -> return vendor ID
213         cpuid
214         movl %eax,X86_CPUID         # save CPUID level
215         movl %ebx,X86_VENDOR_ID         # lo 4 chars
216         movl %edx,X86_VENDOR_ID+4       # next 4 chars
217         movl %ecx,X86_VENDOR_ID+8       # last 4 chars
218
219         orl %eax,%eax               # do we have processor info as well?
220         je is486
221
222         movl $1,%eax            # Use the CPUID instruction to get CPU type
223         cpuid
224         movb %al,%cl            # save reg for future use
225         andb $0x0f,%ah          # mask processor family
226         movb %ah,X86
227         andb $0xf0,%al          # mask model
```

```
228         shrb $4,%al
229         movb %al,X86_MODEL
230         andb $0x0f,%cl       # mask mask revision
231         movb %cl,X86_MASK
232         movl %edx,X86_CAPABILITY
233
234     is486:
235         movl %cr0,%eax       # 486 or better
236         andl $0x80000011,%eax    # Save PG,PE,ET
237         orl $0x50022,%eax    # set AM, WP, NE and MP
238         jmp 2f
239
240     is386:  pushl %ecx       # restore original EFLAGS
241         popfl
242         movl %cr0,%eax       # 386
243         andl $0x80000011,%eax    # Save PG,PE,ET
244         orl $2,%eax      # set MP
245     2:  movl %eax,%cr0
246         call check_x87
```

读者知道，i386 是一个 CPU 芯片系列，其中包括了 80386、80486 以及后来的各种 Pentum 芯片。为了让在具体 CPU 芯片上运行的软件知道是在什么样的 CPU 上运行，在各种 CPU 芯片中都提供了一些手段，让软件可以在运行中加以测试。可是，Intel 并没有从一开始就有一个全盘的考虑，而只是到了 Pentium 才为此专设了一条指令 cpuid。这条指令在寄存器%eax 中返回 CPU 芯片所属的系列、型号、版本等信息。读者在前一章中还看到，这条指令还起着存储器路障的作用。至于 Pentium 以前的处理器，即 80386 和 80486，则要通过一些特殊的操作才能知道是哪一种处理器。我们在这里就不深入到这些细节中去了。至于 246 行，则是测试与 CPU 配套的浮点协处理器 80387 是否存在。

我们在前面讲过，CPU 在进入 startup_32( )时已经运行于保护模式。既然运行于保护模式，那就有个全局段描述表，而控制寄存器 GDTR 则指向这个全局段描述表。可是，那时候的全局段描述表只是临时的，由引导辅助程序 setup 设置。至此为止，CPU 一直在用这个临时的全局段描述表。现在，则要改成使用内核正式的全局段描述表了。同样，此前的控制寄存器 IDTR 也指向临时的中断描述表，现在也要转到内核正式的中断描述表了。这些事也是每个 CPU 都要做的。

[startup_32( )]

```
247     #ifdef CONFIG_SMP
248         incb ready
249     #endif
250         lgdt gdt_descr
251         lidt idt_descr
252         ljmp $(__KERNEL_CS),$1f
253     1:  movl $(__KERNEL_DS),%eax     # reload all the segment registers
254         movl %eax,%ds        # after changing gdt.
255         movl %eax,%es
256         movl %eax,%fs
```

```
257         movl %eax,%gs
258     #ifdef CONFIG_SMP
259         movl $(__KERNEL DS), %eax
260         movl %eax,%ss           # Reload the stack pointer (segment only)
261     #else
262         lss stack_start,%esp        # Load processor stack
263     #endif
264         xorl %eax,%eax
265         lldt %ax
266         cld         # gcc2 wants the direction flag cleared at all times
```

指令 lgdt 和 lidt 分别设置 CPU 的“全局段描述表寄存器”GDTR 和“中断描述表寄存器”IDTR。实际上装入这些寄存器的是一个 Xgt_desc_struct 数据结构，此数据结构的类型定义见 include/asm-i386/desc.h：

```
 51     struct Xgt_desc struct {
 52         unsigned short size;
 53         unsigned long address __attribute__((packed));
 54     };
```

结构中首先是16位的字段size，表示描述表的大小，然后才是具体描述表的地址。这两个数据结构的空间分配也是在arch/i386/kernel/head.S中定义的：

```
372     idt_descr:
373         .word IDT_ENTRIES*8-1       # idt contains 256 entries
374     SYMBOL_NAME(idt):
375         .long SYMBOL_NAME(idt_table)
376
377         .word 0
378     gdt_descr:
379         .word GDT_ENTRIES*8-1
380     SYMBOL_NAME(gdt):
381         .long SYMBOL_NAME(gdt_table)
```

中断描述表 idt_table 的内容已经在前面设置好，现在又设置了 IDTR，就完成了对中断机制的准备，只差打开中断了。

全局段描述表 gdt_table 的定义也在同一文件中：

```
450     ENTRY(gdt_table)
451         .quad 0x0000000000000000    /* NULL descriptor */
452         .quad 0x0000000000000000    /* not used */
453         .quad 0x00cf9a000000ffff    /* 0x10 kernel 4GB code at 0x00000000 */
454         .quad 0x00cf92000000ffff    /* 0x18 kernel 4GB data at 0x00000000 */
455         .quad 0x00cffa000000ffff    /* 0x23 user   4GB code at 0x00000000 */
456         .quad 0x00cff2000000ffff    /* 0x2b user   4GB data at 0x00000000 */
457         .quad 0x0000000000000000    /* not used */
```

```
458         .quad 0x0000000000000000    /* not used */
```

由于改变了 GDTR 的内容，各个段寄存器的内容也要再装入一遍，虽然它们的内容其实并无改变。至于“局部段描述表”，则 Linux 内核并不使用局部段，所以将 LDTR 设成 0(见 264～265 行)。

至此，初始化的第一阶段已经完成。在 SMP 结构的系统中，主 CPU 和次 CPU 的代码又要分道扬镳了。我们再往下看。

[startup_32( )]

```
267     #ifdef CONFIG_SMP
268         movb ready, %cl
269         cmpb $1,%cl
270         je 1f                   # the first CPU calls start_kernel
271                                 # all other CPUs call initialize_secondary
272         call SYMBOL_NAME(initialize_secondary)
273         jmp L6
274     1:
275     #endif
276         call SYMBOL_NAME(start_kernel)
277     L6:
278         jmp L6                  # main should never return here, but
279                                 # just in case, we know what happens.
```

在 SMP 结构的系统中，用变量 ready 对已经执行了初始化第一阶段的 CPU 进行计数(见 248 行)。主 CPU 是首先执行初始化第一阶段的，所以如果此时 ready 为 1 就表明当前 CPU 是主 CPU，否则就是次 CPU。主 CPU 在完成了初始化第一阶段以后还任重道远，所以在 276 行调用 start_kernel( )继续进行第二阶段的初始化。而次 CPU 则是“大树底下好乘凉”，主 CPU 已经为之作好了准备，因此直接就通过 initialize_secondary( )转入其空转进程，有关的过程读者已经在前一章中看到过了。不过，次 CPU 的运行要到主 CPU 基本上完成了第二阶段的初始化时才会启动，而且主 CPU 在每启动一个次 CPU 以后都会等待其完成初始化。

[startup_32( ) > initialize_secondary( )]

```
468     /*
469      * Everything has been set up for the secondary
470      * CPUs - they just need to reload everything
471      * from the task structure
472      * This function must not return.
473      */
474     void __init initialize_secondary(void)
475     {
476         /*
477          * We don't actually need to load the full TSS,
478          * basically just the stack pointer and the eip.
479          */
480
```

```
481         asm volatile(
482             "movl %0,%%esp\n\t"
483             "jmp *%1"
484             :
485             :"r" (current->thread.esp),"r" (current->thread.eip));
486     }
```

这里，次 CPU 的跳转地址和堆栈指针都是主 CPU 为之准备好的。

表面上看来，从 start_kernel( )或 initialize_secondary( )返回以后就落入了一个无限循环(277～278 行)，但是读者将会看到对这两个函数的调用是不会返回的。对于主 CPU 或者单处理器系统中的 CPU，下面就是系统初始化的第二阶段了。

10.3　系统初始化（第二阶段）

从某种意义上说，函数 start_kernel( )就好像一般可执行程序中的主函数 main( )，系统在进入这个函数之前已经进行了一些最低限度的初始化，为这个函数的执行建立起了一个环境，创造了必要的条件。当然，这个函数还要继续进行内核的初始化，实际上甚至可以说内核的初始化这才真正开始。但是这种初始化与在此之前的初始化毕竟不同，是较高层次上的初始化。这也是为什么从这里开始的代码基本上是 C 代码，而在此之前则都是汇编代码的原因。这个函数的代码在 init/main.c 中：

```
516     /*
517      *  Activate the first processor.
518      */
519
520     asmlinkage void __init start_kernel(void)
521     {
522         char * command_line;
523         unsigned long mempages;
524         extern char saved_command_line[ ];
525     /*
526      * Interrupts are still disabled. Do necessary setups, then
527      * enable them
528      */
529         lock_kernel( );
530         printk(linux_banner);
531         setup_arch(&command_line);
```

这里首先通过 printk( )在屏幕上显示出内核的版本信息，这些信息是在编译时生成的，具体可参考 init/version.c 中的有关内容，此处不加赘述：

```
24     const char *linux_banner =
25         "Linux version " UTS_RELEASE " (" LINUX_COMPILE_BY "@"
26         LINUX_COMPILE_HOST ") (" LINUX_COMPILER ") " UTS_VERSION "\n";
```

然后是 setup_arch( )。顾名思义，这个函数所处理的是系统结构的设置，这就是初始化第二阶段的主体。不过我们在这里并不关心为一些特殊要求和情况所作的考虑，所以从函数中抽去了一些条件编译的代码。例如，CONFIG_VISWS 是为 SUN 公司的工作站而设的，CONFIG_BLK_DEV_RAM 是为 RAMDISK 而设的，这些都不在我们关心之列，有兴趣或需要的读者可以自行阅读有关代码。函数 setup_arch( )的代码在 arch/i386/kernel/setup.c 中，我们分段阅读。

[start_kernel( ) > setup_arch( )]

```
598     void __init setup_arch(char **cmdline_p)
599     {
600         unsigned long bootmap_size;
601         unsigned long start_pfn, max_pfn, max_low_pfn;
602         int i;
603
604     #ifdef CONFIG_VISWS
        . . . . . .
606     #endif
607
608         ROOT_DEV = to_kdev_t(ORIG_ROOT_DEV);
609         drive_info = DRIVE_INFO;
610         screen_info = SCREEN_INFO;
611         apm_info.bios = APM_BIOS_INFO;
612         if( SYS_DESC_TABLE.length != 0 ) {
613             MCA_bus = SYS_DESC_TABLE.table[3] &0x2;
614             machine_id = SYS_DESC_TABLE.table[0];
615             machine_submodel_id = SYS_DESC_TABLE.table[1];
616             BIOS_revision = SYS_DESC_TABLE.table[2];
617         }
618         aux_device_present = AUX_DEVICE_INFO;
619
620     #ifdef CONFIG_BLK_DEV_RAM
        . . . . . .
624     #endif
625         setup_memory_region( );
626
```

代码中的 ROOT_DEV 是个全局量，显而易见是根设备的设备号，而 ORIG_ROOT_DEV 就是前述参数块中的一个 16 位设备号，代表着从中引导内核映象的设备，由引导辅助程序 setup 在引导时加以设置并传递给内核。这里先假定引导设备就是根设备，如果在引导命令行中另有指定，则在后面处理命令行时再加修正。其他的 drive_info、screen_info 等也与此类似，有关的信息均来自参数块，实际上是来自 BIOS。其中 MCA_bus 表示系统中是否配备了 PS/2 的 Micro Channel 总线，而 machine_id 和 machine_submodel_id 则显然是具体 PC 机的标号。

如前所述，BIOS 的功能并不只是引导操作系统内核，还担负着加电以后的自检和对资源的扫描探测，其中就包括了对物理内存的自检和扫描(想必读者在开机时看到过 BIOS 在此阶段中显示的信息)，对于在这个阶段中获得的内存信息可以通过 BIOS 调用“int 0x15”加以查询。由于在 Linux 内核中不

能作 BIOS 调用，所以由 setup 在引导阶段代为查询，并根据获得的信息生成一幅物理内存构成图，称为 e820 图，再通过参数块传给内核，使内核能知道系统中内存资源的配置。之所以称为 e820 图，是因为在通过“int 0x15” 查询内存的构成时要把调用参数之一设置成 0xe820。这里的目的是把这个图复制到一个安全的地方，因为参数块只是暂时存放在 empty_zero_page 中，这个页面最后要用于其他目的。

[start_kernel( ) > setup_arch( ) > setup_memory_region( )]

```
491    /*
492     * Do NOT EVER look at the BIOS memory size location.
493     * It does not work on many machines.
494     */
495    #define LOWMEMSIZE( )    (0x9f000)
496
497    void __init setup_memory_region(void)
498    {
499        char *who = "BIOS-e820";
500
501        /*
502         * Try to copy the BIOS-supplied E820-map.
503         *
504         * Otherwise fake a memory map; one section from 0k->640k,
505         * the next section from 1mb->appropriate_mem_k
506         */
507        if (copy_e820_map(E820_MAP, E820_MAP_NR) < 0) {
508            unsigned long mem_size;
509
510            /* compare results from other methods and take the greater */
511            if (ALT_MEM_K < EXT_MEM_K) {
512                mem_size = EXT_MEM_K;
513                who = "BIOS-88";
514            } else {
515                mem_size = ALT_MEM_K;
516                who = "BIOS-e801";
517            }
518
519            e820.nr_map = 0;
520            add_memory_region(0, LOWMEMSIZE( ), E820_RAM);
521           add_memory_region(HIGH_MEMORY, (mem_size << 10) - HIGH_MEMORY, E820_RAM);
522        }
523        printk("BIOS-provided physical RAM map:\n");
524        print_memory_map(who);
525    } /* setup_memory_region */
```

为方便阅读，我们把参数块中与此有关的常数的定义列出于下(见include/asm-i386/e820.h 和 arch/i386/kernel\setup.c)：

```
15      #define E820MAP       0x2d0          /* our map */
16      #define E820MAX       32             /* number of entries in E820MAP */
17      #define E820NR        0x1e8          /* # entries in E820MAP */

158     #define E820_MAP_NR (*(char*) (PARAM+E820NR))
159     #define E820_MAP     ((struct e820entry *) (PARAM+E820MAP))
```

其中，E820_MAP 是个 e820entry 数据结构指针，存放在参数块中位移为 0x2d0 的地方，这种数据结构的定义在 include/asm-i386/e820.h 中：

```
28      struct e820map {
29          int nr_map;
30          struct e820entry {
31          unsigned long long addr;    /* start of memory segment */
32          unsigned long long size;    /* size of memory segment */
33          unsigned long type;     /* type of memory segment */
34          } map[E820MAX];
35      };
36
37      extern struct e820map e820;
```

由此可见，所谓“map”是一个 e820entry 结构数组，数组中的每一项都是对一个物理内存区间的描述。代码中先通过 copy_e820_map( )复制，如果发现数组中的信息可疑，就放弃复制而根据参数块中的其他信息生成(猜测)出有关的内容(508～521 行)。函数 copy_e820_map( )的代码也在 setup.c 中：

[start_kernel( ) > setup_arch( ) > setup_memory_region( ) > copy_e820_map( )]

```
440     /*
441      * Copy the BIOS e820 map into a safe place.
442      *
443      * Sanity-check it while we're at it..
444      *
445      * If we're lucky and live on a modern system, the setup code
446      * will have given us a memory map that we can use to properly
447      * set up memory.  If we aren't, we'll fake a memory map.
448      *
449      * We check to see that the memory map contains at least 2 elements
450      * before we'll use it, because the detection code in setup.S may
451      * not be perfect and most every PC known to man has two memory
452      * regions: one from 0 to 640k, and one from 1mb up.  (The IBM
453      * thinkpad 560x, for example, does not cooperate with the memory
454      * detection code.)
455      */
456     static int __init copy_e820_map(struct e820entry * biosmap, int nr_map)
457     {
458         /* Only one memory region (or negative)? Ignore it */
459         if (nr_map < 2)
```

```
460             return -1;
461
462         do {
463             unsigned long long start = biosmap->addr;
464             unsigned long long size = biosmap->size;
465             unsigned long long end = start + size;
466             unsigned long type = biosmap->type;
467
468             /* Overflow in 64 bits? Ignore the memory map. */
469             if (start > end)
470                 return -1;
471
472             /*
473              * Some BIOSes claim RAM in the 640k - 1M region.
474              * Not right. Fix it up.
475              */
476             if (type == E820_RAM) {
477                 if (start < 0x100000ULL && end > 0xA0000ULL) {
478                     if (start < 0xA0000ULL)
479                         add_memory_region(start, 0xA0000ULL-start, type);
480                     if (end <= 0x100000ULL)
481                         continue;
482                     start = 0x100000ULL;
483                     size = end - start;
484                 }
485             }
486             add_memory_region(start, size, type);
487         } while (biosmap++,--nr_map);
488         return 0;
489     }
```

如上所述，每个 e820entry 结构都是对一个物理内存区间的描述。从数据结构的定义中也可以看出，一个物理内存区间必须是同一类型的。如果有一片地址连续的物理内存空间，其一部分是 RAM，而另一部分是 ROM，那就要分成两个区间。即使同属 RAM，如果其中一部分要保留用于特殊目的，那也属于一个不同的分区。文件 include/asm-i386/e820.h 中定义了 4 种不同的类型：

```
19      #define E820_RAM            1
20      #define E820_RESERVED       2
21      #define E820_ACPI           3 /* usable as RAM once ACPI tables have been read */
22      #define E820_NVS            4
```

其中 E820_NVS 表示“Non-Volatile Storage”，即“不挥发”存储器，包括 ROM、EPROM、Flash 存储器等等。

在 PC 机中，对于最初 1MB 存储空间的使用是特殊的。开头 640KB(即 0x0-0x9FFFF)为 RAM。从 0xA0000 开始的空间则用于 CGA、EGA、VGA 等图形卡。现在已经很少有人用 EGA 或 VGA（更不用提 CGA）了，但是不管是什么图形卡，开机时总是工作于 EGA 或 VGA 模式。从 0xF0000 开始到 0xFFFFF，

即最高的 64KB，那就是在 EPROM 或 Flash 存储器中的 BIOS。所以，只要有 BIOS 存在，就至少也得有两个区间，如果 nr_map 小于 2 就一定错了。此外，如果一个区间的起点地址加上长度以后反而小了，那就说明发生了溢出，一个 64 位的地址(类型为 unsigned long long)发生溢出当然一定是错了。由于 BIOS 和图形卡存储空间的存在，本来可以是连续、均质的 RAM 空间就不连续、不均匀了。当然，现在已经不会再有人会糊涂到有意地设计出这么一种存储空间结构了。之所以会有这样的结构，是因为在 PC 的早期人们觉得“个人计算机”有 640KB 的 RAM 空间已经是匪夷所思，而且当时 Intel X86 系列 CPU 的总的寻址能力也只有 1MB（当时有一种很流行的处理器 Z80，只有 64KB 的寻址能力）。技术的发展往往会使人“大跌眼镜”，使原来合理而且权威的设计变得可笑。后来，1MB 的边界很快就被冲破了，于是把 1MB 以上的空间称为“HIGH_MEMORY”。这个称呼一直沿用到了现在，代码中的常数 HIGH_MEMORY 就定义为(1024×1024)。现在，配备了 128MB RAM 的 PC 机已经是很普通的了。但是，作为一种系统结构，在最初 1MB 的 RAM 空间中还得留下这么个空洞，否则便不能与业已存在的硬件和软件兼容。所以，代码中对于每个区间都调用 add_memory_region( )，将其参数复制到数据结构 e820 内的数组中。而关键在于：当一个 RAM 区间的起点在 0xA0000 以下，而终点在 1MB 以上时，就要将这个区间拆开成两个区间，中间跳过从 0xA0000 到 1MB 边界之间的那一部分。不过，在特殊的情况下也可以通过引导命令行中的选择项改变这种空间结构。

回到 setup_arch( )的代码中，继续往下看。

[start_kernel( ) > setup_arch( )]

```
627         if (!MOUNT_ROOT_RDONLY)
628             root_mountflags &= ~MS_RDONLY;
629         init_mm.start_code = (unsigned long) & text;
630         init_mm.end_code = (unsigned long) & etext;
631         init_mm.end_data = (unsigned long) & edata;
632         init_mm.brk = (unsigned long) &_end;
633 
634         code_resource.start = virt_to_bus(&_text);
635         code_resource.end = virt_to_bus(&_etext)-1;
636         data_resource.start = virt_to_bus(&_etext);
637         data_resource.end = virt_to_bus(&_edata)-1;
638 
639         parse_mem_cmdline(cmdline_p);
640 
641     #define PFN_UP(x)    (((x) + PAGE_SIZE-1) >> PAGE_SHIFT)
642     #define PFN_DOWN(x)  ((x) >> PAGE_SHIFT)
643     #define PFN_PHYS(x)  ((x) << PAGE_SHIFT)
644 
645     /*
646      * 128MB for vmalloc and initrd
647      */
648     #define VMALLOC_RESERVE (unsigned long)(128 << 20)
649     #define MAXMEM          (unsigned long)(-PAGE_OFFSET-VMALLOC_RESERVE)
650     #define MAXMEM_PFN      PFN_DOWN(MAXMEM)
651     #define MAX_NONPAE_PFN  (1 << 20)
652 
```

```
653         /*
654          * partially used pages are not usable - thus
655          * we are rounding upwards:
656          */
657         start_pfn = PFN_UP(__pa(&_end));
658
```

除了对数据结构 init_mm 和 code_resource 的设置以外，这里主要是对 parse_mem_cmdline( )的调用。数据结构 init_mm 是系统中第一个进程 swapper 的存储空间控制结构，也是整个内核的 mm_struct 数据结构。这个数据结构代表着系统空间，不管是什么进程，只要一进入内核就进入了系统空间，就受这个数据结构控制。

在特殊的情况下，有的系统可能有特殊的 RAM 空间结构，此时可以通过引导命令行中的选择项改变存储空间的逻辑结构，使其正确地反映内存的物理结构。函数 parse_mem_cmdline( )的作用就是分析命令行中的选择项，并据此对数据结构 e820 中的内容作出修正。这个函数的代码在 arch/i386/kernel/setup.c 中，但是我们不深入进去了，只是把代码中的一段注释抄录于下，让读者对这些选择项有个大致的印象：

```
539             /*
540              * "mem=nopentium" disables the 4MB page tables.
541              * "mem=XXX[kKmM]" defines a memory region from HIGH_MEM
542              * to <mem>, overriding the bios size.
543              * "mem=XXX[KkmM]@XXX[KkmM]" defines a memory region from
544              * <start> to <start>+<mem>, overriding the bios size.
545              */
```

其中的第一项用于 CPU 为 Pentium，并且内核在编译时采用了 36 位地址(PAE 模式)，又采用了 Pentium 的 PSE 功能，即采用 4MB 页面，但是在引导时却临时决定要采用 4KB 页面的情景。我们在本书中只关心 32 位地址、页面大小为 4KB 的模式，有兴趣或有需要的读者自可对 PAE 和 PSE 加以研究。

然后，代码中就地定义了一些宏操作和常数，这些定义都是下面要用到的。首先就是用在对 start_pfn 的计算（pfn 大概是“Page Frame Number”的缩写），这个变量的值是个页面号，代表着内存中内核映象以上第一个可以动态分配的页面。内核映象的终点是_end，这是由 gcc 在编译和连接时自动生成的，从它的地址往上就是可以动态分配的空间了，宏操作 PFN_UP( )根据这个地址计算出它上面的第一个页面边界。

继续看 setup_arch( )的代码。

[start_kernel( ) > setup_arch( )]

```
659         /*
660          * Find the highest page frame number we have available
661          */
662         max_pfn = 0;
663         for (i = 0; i < e820.nr_map; i++) {
664             unsigned long start, end;
665             /* RAM? */
```

```
666             if (e820.map[i].type != E820_RAM)
667                 continue;
668             start = PFN_UP(e820.map[i].addr);
669             end = PFN_DOWN(e820.map[i].addr + e820.map[i].size);
670             if (start >= end)
671                 continue;
672             if (end > max_pfn)
673                 max_pfn = end;
674         }
675
676         /*
677          * Determine low and high memory ranges:
678          */
679         max_low_pfn = max_pfn;
680         if (max_low_pfn > MAXMEM_PFN) {
681             max_low_pfn = MAXMEM_PFN;
682     #ifndef CONFIG_HIGHMEM
683             /* Maximum memory usable is what is directly addressable */
684             printk(KERN_WARNING "Warning only %ldMB will be used.\n",
685                         MAXMEM>>20);
686             if (max_pfn > MAX_NONPAE_PFN)
687                 printk(KERN_WARNING "Use a PAE enabled kernel.\n");
688             else
689                 printk(KERN_WARNING "Use a HIGHMEM enabled kernel.\n");
690     #else /* !CONFIG_HIGHMEM */
691     #ifndef CONFIG_X86_PAE
692             if (max_pfn > MAX_NONPAE_PFN) {
693                 max_pfn = MAX_NONPAE_PFN;
694                 printk(KERN_WARNING "Warning only 4GB will be used.\n");
695                 printk(KERN_WARNING "Use a PAE enabled kernel.\n");
696             }
697     #endif /* !CONFIG_X86_PAE */
698     #endif /* !CONFIG_HIGHMEM */
699         }
700
701     #ifdef CONFIG_HIGHMEM
702         highstart_pfn = highend_pfn = max_pfn;
703         if (max_pfn > MAXMEM_PFN) {
704             highstart_pfn = MAXMEM_PFN;
705             printk(KERN_NOTICE "%ldMB HIGHMEM available.\n",
706                 pages_to_mb(highend_pfn - highstart_pfn));
707         }
708     #endif
```

在数据结构e820中积累起各个物理内存区间的信息以后，要从这些信息中归纳出一项重要的数据，那就是RAM空间的顶点max_pfn。代码中通过一个for循环扫描e820中的数组，通过比较和计算得出此项数据。宏操作PFN_UP( )和PFN_DOWN( )的定义见上面的641和642行，作用是将内存地址转换

成页面号，因此 max_pfn 所代表的是系统中最高的 RAM 页面。不过，实际可以使用多大的物理内存不光取决于物理上的配备以及整个地址空间的大小，还进一步受到内核的虚存空间，即所谓系统空间大小的限制。读者在第 2 章中曾看到，Linux 的内核将整个 32 位地址空间分成两部分，其中 0xC0000000 以上的 1GB 为系统空间，从系统空间到物理内存的映射基本上是线性映射。如果物理内存的大小超过了 1GB，就不能从内核中访问到整个物理内存。另一方面，虽然从系统空间到物理内存的映射基本上是线性映射、因而不会将同一物理内存页面重复地映射到系统空间中，但也还是有例外的。我们在第 2 章中提到过 vmalloc( )，它就是在原有的线性映射以外另行分配一块系统空间（虚拟地址），并建立起与物理页面的映射。此外，如果采用 RAMDISK，则也有类似的情况。这样，由于同一物理页面有可能耗用一个以上的虚存页面，可以在系统空间中直接（不需要通过临时改变映射）访问的物理页面数量就又减少了，不过这一部分虚存空间的大小限于 128MB。因此，从可以被内核直接访问的角度，理论上最大的 RAM 空间容量为 1024MB－128MB = 896MB，这就是常数 MAXMEM ，相应的页面号为 MAXMEM_PFN。这个数值有可能小于 max_pfn，所以另有一个变量 max_low_pfn 用来记录这个数值。下面是几个有关常数的定义，分别见于 arch/i386/kernel/setup.c 和 include/asm-i386/page.h：

```
645    /*
646     * 128MB for vmalloc and initrd
647     */
648    #define VMALLOC_RESERVE      (unsigned long)(128 << 20)
649    #define MAXMEM               (unsigned long)(-PAGE_OFFSET-VMALLOC_RESERVE)
650    #define MAXMEM_PFN           PFN_DOWN(MAXMEM)

 81    #define __PAGE_OFFSET        (0xC0000000)
```

所以，对于 Linux 内核，如果系统中配备了 896MB 以上的 RAM，就需要超过 32 位的寻址能力，此时要选用编译选择项 CONFIG_HIGHMEM，注意这里所谓 HIGHMEM 是指 4GB 虚存空间，而不是前面讲的 1MB，不要搞混淆了。从代码中的 682～698 行可以看出，如果配备了 896MB 以上的 RAM，而又不选用 CONFIG_HIGHMEM，则只能使用其中的 896MB。如果选用了 CONFIG_HIGHMEM，则又要看是否选用 Pentium 的 PAE 模式，那些我们就不关心了，反正现在大概还没有多少人要用 896MB 以上的物理内存(几年以后也许又要大跌眼镜)。还要指出，这只是对 32 位结构的 i386 而言，对新的 ia64 结构当然要另作别论。

我们再往下看。

[start_kernel( ) > setup_arch( )]

```
709        /*
710         * Initialize the boot-time allocator (with low memory only):
711         */
712        bootmap_size = init_bootmem(start_pfn, max_low_pfn);
713
714        /*
715         * Register fully available low RAM pages with the bootmem allocator.
716         */
717        for (i = 0; i < e820.nr_map; i++) {
718            unsigned long curr_pfn, last_pfn, size;
```

```
719             /*
720              * Reserve usable low memory
721              */
722             if (e820.map[i].type != E820_RAM)
723                 continue;
724             /*
725              * We are rounding up the start address of usable memory:
726              */
727             curr_pfn = PFN_UP(e820.map[i].addr);
728             if (curr_pfn >= max_low_pfn)
729                 continue;
730             /*
731              * ... and at the end of the usable range downwards:
732              */
733             last_pfn = PFN_DOWN(e820.map[i].addr + e820.map[i].size);
734 
735             if (last_pfn > max_low_pfn)
736                 last_pfn = max_low_pfn;
737 
738             /*
739              * .. finally, did all the rounding and playing
740              * around just make the area go away?
741              */
742             if (last_pfn <= curr_pfn)
743                 continue;
744 
745             size = last_pfn - curr_pfn;
746             free_bootmem(PFN_PHYS(curr_pfn), PFN_PHYS(size));
747         }
748         /*
749          * Reserve the bootmem bitmap itself as well. We do this in two
750          * steps (first step was init_bootmem( )) because this catches
751          * the (very unlikely) case of us accidentally initializing the
752          * bootmem allocator with an invalid RAM area.
753          */
754         reserve_bootmem(HIGH_MEMORY, (PFN_PHYS(start_pfn) +
755                 bootmap_size + PAGE_SIZE-1) - (HIGH_MEMORY));
756 
757         /*
758          * reserve physical page 0 - it's a special BIOS page on many boxes,
759          * enabling clean reboots, SMP operation, laptop functions.
760          */
761         reserve_bootmem(0, PAGE_SIZE);
762 
763     #ifdef CONFIG_SMP
764         /*
765          * But first pinch a few for the stack/trampoline stuff
766          * FIXME: Don't need the extra page at 4K, but need to fix
```

```
767          * trampoline before removing it. (see the GDT stuff)
768          */
769         reserve_bootmem(PAGE_SIZE, PAGE_SIZE);
770         smp_alloc_memory( ); /* AP processor realmode stacks in low memory*/
771     #endif
772 
773     #ifdef CONFIG_X86_IO_APIC
774         /*
775          * Find and reserve possible boot-time SMP configuration:
776          */
777         find_smp_config( );
778     #endif
779         paging_init( );
```

首先通过 init_bootmem( )为物理内存页面管理机制的建立作些准备，为整个物理内存建立起一个页面位图。这个位图建立在从 start_pfn 开始的地方。也就是说，把内核映象终点_end 上方的若干页面用作物理页面位图。在此之前的代码中已经搞清楚了物理内存顶点所在的页面号是 max_low_pfn，所以整个物理内存的页面号一定在 0 至 max_low_pfn 这个范围内。可是，在这个范围中可能有空洞，另一方面也并不是所有的物理内存页面都可以动态分配。建立这个位图的目的就是要搞清楚哪一些物理内存页面是可以动态分配的。

[start_kernel( ) > setup_arch( ) > init_bootmem( )]

```
283     unsigned long __init init_bootmem (unsigned long start, unsigned long pages)
284     {
285         max_low_pfn = pages;
286         min_low_pfn = start;
287         return(init_bootmem_core(&contig_page_data, start, 0, pages));
288     }
```

操作的主体是 init_bootmem_core( )，这个函数对 pg_data_t 数据结构 contig_page_data 进行初始化，读者已经在第 2 章的“几个重要的数据结构和函数”一节中看到过这种数据结构的定义。每个 pg_data_t 数据结构都代表着一片均匀的、连续的内存空间，称为一个“节点”。在连续空间 UMA 结构中只有一个节点 contig_page_data，而在 NUMA 结构或不连续空间 UMA 结构中则有多个这样的数据结构。系统中各个节点的 pg_data_t 数据结构通过指针 node_next 连接在一起成为一个链，全局量 pgdat_list 则指向这个链。显然，contig_page_data 是链中的第一个节点。这里先假定整个物理内存空间为均匀的、连续的，以后若发现这个假定不能成立则再加以修正，再将新的 pg_data_t 结构加入到链中。数据结构 contig_page_data 的初始内容为(mm/numa.c)：

```
14      static bootmem_data_t contig_bootmem_data;
15      pg_data_t contig_page_data = { bdata: &contig_bootmem_data };
```

在 pg_data_t 结构中有个指针 bdata，指向一个 bootmem_data_t 数据结构，其类型定义如下(include/linux/bootmem.h)：

```
20    /*
21     * node_bootmem_map is a map pointer - the bits represent all physical
22     * memory pages (including holes) on the node.
23     */
24    typedef struct bootmem_data {
25        unsigned long node_boot_start;
26        unsigned long node_low_pfn;
27        void *node_bootmem_map;
28        unsigned long last_offset;
29        unsigned long last_pos;
30    } bootmem_data_t;
```

结构中的 node_boot_start 表示系统引导以后存在的第一个物理内存页面(而与引导过程本身无关)，从调用 init_bootmem_core( )时的参数 start 可以看出是 0；node_low_pfn 则表示物理内存的顶点，最高不超过 896MB。结构中的指针 node_bootmem_map 指向一个“保留页面位图”，位图中的每一位都代表着物理内存中一个需要保留，或者不存在，从而不能用于动态分配的页面。函数 init_bootmem_core( ) 的代码在 mm/bootmem.c 中：

[start_kernel( ) > setup_arch( ) > init_bootmem( ) > init_bootmem_core( )]

```
41    /*
42     * Called once to set up the allocator itself.
43     */
44    static unsigned long __init init_bootmem_core (pg_data_t *pgdat,
45        unsigned long mapstart, unsigned long start, unsigned long end)
46    {
47        bootmem_data_t *bdata = pgdat->bdata;
48        unsigned long mapsize = ((end - start)+7)/8;
49
50        pgdat->node_next = pgdat_list;
51        pgdat_list = pgdat;
52
53        mapsize = (mapsize + (sizeof(long) - 1UL)) & ~(sizeof(long) - 1UL);
54        bdata->node_bootmem_map = phys_to_virt(mapstart << PAGE_SHIFT);
55        bdata->node_boot_start = (start << PAGE_SHIFT);
56        bdata->node_low_pfn = end;
57
58        /*
59         * Initially all pages are reserved - setup_arch( ) has to
60         * register free RAM areas explicitly.
61         */
62        memset(bdata->node_bootmem_map, 0xff, mapsize);
63
64        return mapsize;
65    }
```

注意这里参数 start 的值为 0，而 mapstart 的值才是上一层函数 init_bootmem( )中的 start，即内核映

象以上第一个页面的起点，参数 end 的值则为物理内存的顶点 max_low_pfn。

我们基本上把这个函数以及 setup_arch( )中的 717～761 行留给读者自己结合着阅读。这里只作简略的说明：

函数 free_bootmem( )对一个 contig_page_data 结构中的位图进行操作，将其中的某些位清 0，表示相应的物理内存页面可以投入分配。而 reserve_bootmem( )则正好相反，一开始时把位图中的所有位都设置成 1，假定全部都不能用于动态分配，然后根据 e820 数据结构中的内容以及一些特殊的区间和页面加以找补。其中特别加以保留的有：

- 从 HIGH_MEMORY，即 1MB 边界开始，直到(start_pfn + bootmap_size)所在的页面为止。这些是内核映象和"保留页面位图"本身所在的页面。
- 页面 0，即起始地址为 0 的页面。BIOS 通常用这个页面保存一些与引导以及 BIOS 本身有关的信息，所以也要加以保留。
- 对于 SMP 结构的系统，页面 1，即起始地址为 PAGE_SIZE 的页面也要保留，次 CPU 转入运行时需要用这个页面作为"跳板"(见第 9 章)。
- 此外，还有一些特殊用途的页面，例如用作 RAMDISK 的页面。不过这些页面不是在这儿保留的。

对于采用 CONFIG_X86_IO_APIC 的 SMP 系统，还要通过 find_smp_config( )寻找由 BIOS 和引导装入程序设置在基本内存(640KB)中的多处理器配置表，不过我们在这里不深入到这个函数中去了。

前面已经建立了为内存页面管理所需的数据结构，现在是进一步完善页面映射机制，并且建立起内存页面管理机制的时候了(779行)。函数paging_init( )的代码在mm/init.c中：

[start_kernel( ) > setup_arch( ) > paging_init( )]

```
437    /*
438     * paging_init( ) sets up the page tables - note that the first 8MB are
439     * already mapped by head.S.
440     *
441     * This routines also unmaps the page at virtual kernel address 0, so
442     * that we can trap those pesky NULL-reference errors in the kernel.
443     */
444    void __init paging_init(void)
445    {
446        pagetable_init( );
447
448        __asm__( "movl %%ecx,%%cr3\n" ::"c"(__pa(swapper_pg_dir)));
449
450    #if CONFIG_X86_PAE
451        /*
452         * We will bail out later - printk doesnt work right now so
453         * the user would just see a hanging kernel.
454         */
455        if (cpu_has_pae)
456            set_in_cr4(X86_CR4_PAE);
457    #endif
458
```

```
459         __flush_tlb_all( );
460
461     #ifdef CONFIG_HIGHMEM
462         kmap_init( );
463     #endif
464         {
465             unsigned long zones_size[MAX_NR_ZONES] = {0, 0, 0};
466             unsigned int max_dma, high, low;
467
468             max_dma = virt_to_phys((char *)MAX_DMA_ADDRESS) >> PAGE_SHIFT;
469             low = max_low_pfn;
470             high = highend_pfn;
471
472             if (low < max_dma)
473                 zones_size[ZONE_DMA] = low;
474             else {
475                 zones_size[ZONE_DMA] = max_dma;
476                 zones_size[ZONE_NORMAL] = low - max_dma;
477     #ifdef CONFIG_HIGHMEM
478                 zones_size[ZONE_HIGHMEM] = high - low;
479     #endif
480             }
481             free_area_init(zones_size);
482         }
483         return;
484     }
```

首先通过 pagetable_init( )扩充由 startup_32( )在第一阶段中创建的页面映射目录和页面映射表（见 arch/i386/kernel/head.S 中的 389 行和 98～102 行）。当初因为不知道内存到底有多大，所以只为开始的 8MB 建立了映射。现在既然已经有了关于内存的详细信息，就可以根据这些信息加以扩充和修改，建立完整的从系统空间到整个物理存储空间的线性映射了。函数 pagetable_init( )的代码也在同一文件中：

[start_kernel( ) > setup_arch( ) > paging_init( ) > pagetable_init( )]

```
314     static void __init pagetable_init (void)
315     {
316         unsigned long vaddr, end;
317         pgd_t *pgd, *pgd_base;
318         int i, j, k;
319         pmd_t *pmd;
320         pte_t *pte;
321
322         /*
323          * This can be zero as well - no problem, in that case we exit
324          * the loops anyway due to the PTRS_PER_* conditions.
325          */
326         end = (unsigned long)__va(max_low_pfn*PAGE_SIZE);
```

```
327
328         pgd_base = swapper_pg_dir;
329     #if CONFIG_X86_PAE
330         for (i = 0; i < PTRS_PER_PGD; i++) {
331             pgd = pgd_base + i;
332             __pgd_clear(pgd);
333         }
334     #endif
335         i = __pgd_offset(PAGE_OFFSET);
336         pgd = pgd_base + i;
337
338         for (; i < PTRS_PER_PGD; pgd++, i++) {
339             vaddr = i*PGDIR_SIZE;
340             if (end && (vaddr >= end))
341                 break;
342     #if CONFIG_X86_PAE
343             pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
344             set_pgd(pgd, __pgd(__pa(pmd) + 0x1));
345     #else
346             pmd = (pmd_t *)pgd;
347     #endif
348             if (pmd != pmd_offset(pgd, 0))
349                 BUG( );
350             for (j = 0; j < PTRS_PER_PMD; pmd++, j++) {
351                 vaddr = i*PGDIR_SIZE + j*PMD_SIZE;
352                 if (end && (vaddr >= end))
353                     break;
354                 if (cpu_has_pse) {
355                     unsigned long __pe;
356
357                     set_in_cr4(X86_CR4_PSE);
358                     boot_cpu_data.wp_works_ok = 1;
359                     __pe = _KERNPG_TABLE + _PAGE_PSE + __pa(vaddr);
360                     /* Make it "global" too if supported */
361                     if (cpu_has_pge) {
362                         set_in_cr4(X86_CR4_PGE);
363                         __pe += _PAGE_GLOBAL;
364                     }
365                     set_pmd(pmd, __pmd(__pe));
366                     continue;
367                 }
368
369                 pte = (pte_t *) alloc_bootmem_low_pages(PAGE_SIZE);
370                 set_pmd(pmd, __pmd(_KERNPG_TABLE + __pa(pte)));
371
372                 if (pte != pte_offset(pmd, 0))
373                     BUG( );
374
```

```
375                 for (k = 0; k < PTRS_PER_PTE; pte++, k++) {
376                     vaddr = i*PGDIR_SIZE + j*PMD_SIZE + k*PAGE_SIZE;
377                     if (end && (vaddr >= end))
378                         break;
379                     *pte = mk_pte_phys(__pa(vaddr), PAGE_KERNEL);
380                 }
381             }
382         }
383 
384         /*
385          * Fixed mappings, only the page table structure has to be
386          * created - mappings will be set by set_fixmap( ):
387          */
388         vaddr = __fix_to_virt(__end_of_fixed_addresses - 1) & PMD_MASK;
389         fixrange_init(vaddr, 0, pgd_base);
390 
391     #if CONFIG_HIGHMEM
392         /*
393          * Permanent kmaps:
394          */
395         vaddr = PKMAP_BASE;
396         fixrange_init(vaddr, vaddr + PAGE_SIZE*LAST_PKMAP, pgd_base);
397 
398         pgd = swapper_pg_dir + __pgd_offset(vaddr);
399         pmd = pmd_offset(pgd, vaddr);
400         pte = pte_offset(pmd, vaddr);
401         pkmap_page_table = pte;
402     #endif
403 
404     #if CONFIG_X86_PAE
405         /*
406          * Add low memory identity-mappings - SMP needs it when
407          * starting up on an AP from real-mode. In the non-PAE
408          * case we already have these mappings through head.S.
409          * All user-space mappings are explicitly cleared after
410          * SMP startup.
411          */
412         pgd_base[0] = pgd_base[USER_PTRS_PER_PGD];
413     #endif
414     }
```

我们把这段代码留给读者作为对第 2 章中有关内容的复习。注意，这里对页面映射目录swapper_pg_dir 中目录项的设置是从下标__pgd_offset(PAGE_OFFSET)开始，即从虚拟地址 0xC0000000 中的最高 10 位 0x300 开始。页面映射目录的大小为一个页面，实际上就是个大小为 1024 的指针数组，这里从下标 0x300，即 768 开始，这是为系统空间准备的。一旦内核线程进入正常运行以后，就不会使用 0xC0000000 以下的虚拟地址了，所以这个映射目录中前 3/4 的目录项最终都要设置成 0，不过现在还不到时候。

设置好页面映射目录以后，448 行的汇编指令 movl 将目录的起始地址 swapper_pg_dir 送入控制寄存器%cr3。读者也许还记得，前面在 startup_32( )中已经把这个地址设置进%cr3 中了(99 行)，为什么这里还要再设置一次呢？这是因为每当设置%cr3 时 CPU 就会将页面映射目录所在的页面装入 CPU 内部高速缓存中的 TLB 部分。现在内存中(实际上是高速缓存中，见下)的映射目录变了，就要再让 CPU 装入一次。由于页面映射机制本来就是开启着的，所以从这条指令以后就扩大了系统空间中有映射区间的大小，使映射的范围覆盖到整个物理内存(HIGHMEM 除外)。不过，这里所谓“内存中”是逻辑意义上的，实际上此时 swapper_pg_dir 中已经改变的目录项很可能还在高速缓存中，所以还要通过 __flush_tlb_all( )将高速缓存中的内容冲刷到内存中，这样才能保证内存中映射目录内容的一致性。

如前所述，要是系统中配备了高于 896MB 以上的内存，而 CPU 又只有 32 位的寻址能力，那就得选用 CONFIG_HIGHMEM 选项。那么，选用了 CONFIG_HIGHMEM 选项以后又是怎样实现对 896MB 以上物理页面的映射呢？根据所谓“抽屉原理”，如果要将多于 n 个的物件(在这里是物理页面)放入 n 个抽屉(在这里是属于系统空间的页面映射表项)，则至少有一个抽屉里要存放多于一个的物件。但是，一个页面表项在同一时间内又确实只能映射一个物理页面，那就说明至少有一个页面表项要在不同的时间里映射到不同的物理页面。对于选用了 CONFIG_HIGHMEM 的系统，内核中设置了一个全局的 pte_t 指针 kmap_pte，指向页面映射表中的一个表项，这个表项将动态地映射到不同的物理页面。每当要访问一个属于“高内存”的物理页面时，就要先改变这个表项。函数 kmap_init( )的作用主要就是设置好指针 kmap_pte。

在第 2 章中，读者曾看到在代表着存储节点的 pg_data_t 数据结构中的一个数组 node_zones[ ]，其大小为 3。通过这个数组，内核将每个节点中的物理页面划分成三个“管理区”(zone)，即 ZONE_DMA、ZONE_NORMAL 以及 ZONE_HIGHMEM，其中 ZONE_HIGHMEM 只有在选用了 CONFIG_HIGHMEM 选项时才有效。为什么要这样划分呢？将 ZONE_HIGHMEM 区分出来是不言而喻的，因为访问这些“高内存”页面时要遵循特殊的步骤。进一步将“低内存”页面也分成两部分，则是因为 DMA 操作对目标页面的特殊要求。在 PC 机中，能够对之进行 DMA 操作的页面必须低于 0x1000000，即 16MB，相应的系统空间虚拟地址为 MAX_DMA_ADDRESS，这个常数在 include/asm-i386/dma.h 中定义为：

```
75      /* The maximum address that we can perform a DMA transfer to on this platform*/
76      #define MAX_DMA_ADDRESS         (PAGE_OFFSET+0x1000000)
```

所以，把 16MB 以下的页面划入 ZONE_DMA 管理区，而从 16MB 以上直至 max_low_pfn 则划入 ZONE_NORMAL 管理区。代码中的 468～479 行准备下一个整数数组 zones_size[ ]，数组中记录了对物理内存的划分，然后将其传递给函数 free_area_init( )，由这个函数根据数组 zones_size[ ]提供的指示在三个管理区中创建起空闲页面块队列。函数 free_area_init( )及有关的代码在 mm/page_alloc.c 中，我们也把它留给读者。

至此，paging_init( )的操作已经完成，我们回到 setup_arch( )的代码中：

[start_kernel( ) > setup_arch( )]

```
780     #ifdef CONFIG_X86_IO_APIC
781         /*
782          * get boot-time SMP configuration:
783          */
784         if (smp_found_config)
```

```
785             get_smp_config( );
786     #endif
787     #ifdef CONFIG_X86_LOCAL_APIC
788         init_apic_mappings( );
789     #endif
790
791     #ifdef CONFIG_BLK_DEV_INITRD
        . . . . . .
807     #endif
808
809         /*
810          * Request address space for all standard RAM and ROM resources
811          * and also for regions reported as reserved by the e820.
812          */
813         probe_roms( );
814         for (i = 0; i < e820.nr_map; i++) {
815             struct resource *res;
816             if (e820.map[i].addr + e820.map[i].size > 0x100000000ULL)
817                 continue;
818             res = alloc_bootmem_low(sizeof(struct resource));
819             switch (e820.map[i].type) {
820             case E820_RAM:  res->name = "System RAM"; break;
821             case E820_ACPI: res->name = "ACPI Tables"; break;
822             case E820_NVS:  res->name = "ACPI Non-volatile Storage"; break;
823             default:     res->name = "reserved";
824             }
825             res->start = e820.map[i].addr;
826             res->end = res->start + e820.map[i].size - 1;
827             res->flags = IORESOURCE_MEM | IORESOURCE_BUSY;
828             request_resource(&iomem_resource, res);
829             if (e820.map[i].type == E820_RAM) {
830                 /*
831                  *  We dont't know which RAM region contains kernel data,
832                  *  so we try it repeatedly and let the resource manager
833                  *  test it.
834                  */
835                 request_resource(res, &code_resource);
836                 request_resource(res, &data_resource);
837             }
838         }
839         request_resource(&iomem_resource, &vram_resource);
840
841         /* request I/O space for devices used on all i[345]86 PCs */
842         for (i = 0; i < STANDARD_IO_RESOURCES; i++)
843             request_resource(&ioport_resource, standard_io_resources+i);
844
845     #ifdef CONFIG_VT
846     #if defined(CONFIG_VGA_CONSOLE)
```

```
847         conswitchp = &vga_con;
848     #elif defined(CONFIG_DUMMY_CONSOLE)
849         conswitchp = &dummy_con;
850     #endif
851     #endif
852     }
```

物理内存页面当然是重要的资源，对这些资源的管理机制已经建立了。但是，从另一个角度看，地址空间本身，或者物理存储器在地址空间中的位置，也是一种资源，也要加以管理。为此，include/linux/ioport.h 中定义了一种 resource 数据结构：

```
11      /*
12       * Resources are tree-like, allowing
13       * nesting etc..
14       */
15      struct resource {
16          const char *name;
17          unsigned long start, end;
18          unsigned long flags;
19          struct resource *parent, *sibling, *child;
20      };
```

这种数据结构描述的是一片可以通过访问内存操作或 I/O 操作加以访问的连续空间，或者说操作对象在相应空间中的位置。这样的对象有 RAM、ROM 以及一些用于外部设备的器件。这里所关心的只是由这些器件所形成的逻辑意义上的连续存储空间或 I/O 地址空间，而并不关心每个这样的空间在物理上包含了几个芯片。另一方面，我们以前讲过，i386 处理器有独立于内存访问指令的 I/O 指令和独立于内存地址空间的 I/O 地址空间。虽然有些处理器没有独立的 I/O 指令和 I/O 地址空间，但是逻辑上还是能根据用途区分某项资源是属于存储性质的还是 I/O 性质的。每个 resource 结构通过其 parent、sibling、child 等指针互相连接在一起，形成一种树状结构。如果一个 resource 结构是另一个 resource 结构的子节点，则它所描述的空间是父节点所描述空间的一部分，或者是对父节点的具体化。内核中有两棵这样的树，一棵是 iomem_resource，另一棵是 ioport_resource，分别代表着两类不同性质的地址资源。两棵树的根本身也都是 resource 数据结构，不过这两个数据结构所描述的并不是用于具体操作对象的地址资源，而是概念上的整个地址空间(kernel/resource.c)。

```
18      struct resource ioport_resource={"PCI IO", 0x0000, IO_SPACE_LIMIT, IORESOURCE_IO};
19      struct resource iomem_resource ={"PCI mem", 0x00000000, 0xffffffff, IORESOURCE_MEM};
```

内核中设置了一个 resource 结构数组 rom_resources[ ]，用来记录系统中每个 ROM 空间的位置(arch/i386/kernel/setup.c)。所谓 ROM 实际上是指所有的不挥发只读存储器，包括 PROM、EPROM、Flash 存储器等存储器件。不过，这个数组中只包括 PC 机所固有的那些 ROM 空间，原则上都在母板上或第一块图形卡上，而不包括由其他接口卡提供的不挥发存储空间。以前说过，只要是 PC 机就至少有一个 ROM 空间，因为其 BIOS 总是在一个 ROM 空间中，并且所在的位置总是固定的。所以，数组中的第一个元素固定地初始化成“System ROM”，位置是从 0xF0000 到 0xFFFFF。此外，图形接口中也有一

个 ROM 空间，其位置应该是从 0xC0000 到 0xC7fff，不过这是要加以验证的。

```
325     /* System ROM resources */
326     #define MAXROMS 6
327     static struct resource rom_resources[MAXROMS] = {
328         { "System ROM", 0xF0000, 0xFFFFF, IORESOURCE_BUSY },
329         { "Video ROM", 0xc0000, 0xc7fff, IORESOURCE_BUSY }
330     };
```

除 BIOS 所在的 ROM 空间以外，其他的 ROM 空间都要通过扫描来寻找或验证。不过，对 ROM 空间在内存中的位置是有限制的，它们只能出现在几个特定的区间，所以只需要扫描这几个特定的区间就可以了，函数 probe_roms( )的代码在 arch/i386/kernel/setup.c 中：

[start_kernel( ) > setup_arch( ) > probe_roms( )]

```
332     #define romsignature(x) (*(unsigned short *)(x) == 0xaa55)
333
334     static void __init probe_roms(void)
335     {
336         int roms = 1;
337         unsigned long base;
338         unsigned char *romstart;
339
340         request_resource(&iomem_resource, rom_resources+0);
341
342         /* Video ROM is standard at C000:0000 - C7FF:0000, check signature */
343         for (base = 0xC0000; base < 0xE0000; base += 2048) {
344             romstart = bus_to_virt(base);
345             if (!romsignature(romstart))
346                 continue;
347             request_resource(&iomem_resource, rom_resources + roms);
348             roms++;
349             break;
350         }
351
352         /* Extension roms at C800:0000 - DFFF:0000 */
353         for (base = 0xC8000; base < 0xE0000; base += 2048) {
354             unsigned long length;
355
356             romstart = bus_to_virt(base);
357             if (!romsignature(romstart))
358                 continue;
359             length = romstart[2] * 512;
360             if (length) {
361                 unsigned int i;
```

```
362                 unsigned char chksum;
363
364                 chksum = 0;
365                 for (i = 0; i < length; i++)
366                     chksum += romstart[i];
367
368                 /* Good checksum? */
369                 if (!chksum) {
370                     rom_resources[roms].start = base;
371                     rom_resources[roms].end = base + length - 1;
372                     rom_resources[roms].name = "Extension ROM";
373                     rom_resources[roms].flags = IORESOURCE_BUSY;
374
375                     request_resource(&iomem_resource, rom_resources + roms);
376                     roms++;
377                     if (roms >= MAXROMS)
378                         return;
379                 }
380             }
381         }
382
383         /* Final check for motherboard extension rom at E000:0000 */
384         base = 0xE0000;
385         romstart = bus_to_virt(base);
386
387         if (romsignature(romstart)) {
388             rom_resources[roms].start = base;
389             rom_resources[roms].end = base + 65535;
390             rom_resources[roms].name = "Extension ROM";
391             rom_resources[roms].flags = IORESOURCE_BUSY;
392
393             request_resource(&iomem_resource, rom_resources + roms);
394         }
395     }
```

第一项ROM资源，即BIOS所在的ROM空间是固有的，所以不经扫描就直接通过request_resource()把它链入iomem_resource树中。有关的代码在kernel/resource.c中：

[start_kernel() > setup_arch() > probe_roms() > request_resource()]

```
114     int request_resource(struct resource *root, struct resource *new)
115     {
116         struct resource *conflict;
117
118         write_lock(&resource_lock);
119         conflict = __request_resource(root, new);
120         write_unlock(&resource_lock);
```

```
121         return conflict ? -EBUSY : 0;
122     }
```

操作的主体是__request_resource( )，其代码也在 kernel/resource.c 中：

[start_kernel( ) > setup_arch( ) > probe_roms( ) > request_resource( ) > __request_resource( )]

```
66      /* Return the conflict entry if you can't request it */
67      static struct resource * __request_resource(struct resource *root,
                                                    struct resource *new)
68      {
69          unsigned long start = new->start;
70          unsigned long end = new->end;
71          struct resource *tmp, **p;
72
73          if (end < start)
74              return root;
75          if (start < root->start)
76              return root;
77          if (end > root->end)
78              return root;
79          p = &root->child;
80          for (;;) {
81              tmp = *p;
82              if (!tmp || tmp->start > end) {
83                  new->sibling = tmp;
84                  *p = new;
85                  new->parent = root;
86                  return NULL;
87              }
88              p = &tmp->sibling;
89              if (tmp->end < start)
90                  continue;
91              return tmp;
92          }
93      }
```

由于 ROM 空间的位置限制在几个特定的区间，不需考虑与其他资源冲突的可能性，所以在 probe_roms( )中并不检查 request_resource( )的返回值。除 BIOS 所在的 ROM 空间之外，其他的都要通过扫描和验证。每个 ROM 空间的开头两个字节一定是 0xaa55，称为“ROM 签名”，由宏操作 romsignature( )加以检验。有的 ROM 空间只要存在就一定在某个特定的地址上，有的则可以在一个特定的区间中，但一定与 2KB 边界对齐。这种多样性是在长期的发展过程中形成的。

回到 setup_arch( )的代码中，将母板上的 ROM 空间纳入 iomem_resource 树中以后，还要根据前面收集在 e820 数据结构中的信息，将由 BIOS 扫描探测到的各个内存区间也纳入 iomem_resource 树的统一管理(814～838 行)。对于 RAM 空间，这里还进一步分解出内核的代码段和数据段两片特殊用途的空间。此外，图形接口上也有一片 RAM 空间，其地址为 0xA0000～0xBFFFF（见 arch/i386/kernel/setup.c）。

```
321    static struct resource code_resource = { "Kernel code", 0x100000, 0 };
322    static struct resource data_resource = { "Kernel data", 0, 0 };
323    static struct resource  vram_resource = { "Video RAM area",
                                    0xa0000, 0xbffff, IORESOURCE_BUSY };
```

对于内核代码段和数据段中的地址信息，是由 setup_arch( )在一开始时根据由 gcc 提供的_text、_etext、_edata 等信息设置的(见 634~637 行)。

系统固有的 I/O 类资源则定义于数组 standard_io_resources[ ]中(arch/i386/kernel/setup.c)，842~843 行的 for 循环将这些 resource 结构链入 ioport_resource 树中：

```
308    struct resource standard_io_resources[ ] = {
309         { "dma1", 0x00, 0x1f, IORESOURCE_BUSY },
310         { "pic1", 0x20, 0x3f, IORESOURCE_BUSY },
311         { "timer", 0x40, 0x5f, IORESOURCE_BUSY },
312         { "keyboard", 0x60, 0x6f, IORESOURCE_BUSY },
313         { "dma page reg", 0x80, 0x8f, IORESOURCE_BUSY },
314         { "pic2", 0xa0, 0xbf, IORESOURCE_BUSY },
315         { "dma2", 0xc0, 0xdf, IORESOURCE_BUSY },
316         { "fpu", 0xf0, 0xff, IORESOURCE_BUSY }
317    };
```

以结构中的第一项为例，I/O 地址 0x00~0x1f 共 32 个寄存器是用于第一个 DMA 控制器的。这里固定设置好的都是系统固有的资源，使用了 0x00~0xff 共 256 个寄存器。从 0x100 开始就是用于一般外部设备的 I/O 地址，需要在相应外部设备的初始化过程中向系统登记。

最后，一般 PC 机都配备 VGA 图形卡，所以使作为全局量的指针 conswitchp 指向数据结构 vga_con。这是 VGA 卡设备驱动层的函数跳转结构，提供了各种屏幕操作的函数指针。

在系统初始化的第二阶段，setup_arch( )也许是最重要的函数，正如其函数所暗示的那样，它所设置的是系统的“architecture”，即总体的结构。完成了 setup_arch( )，内核就有了一个框架，内核中的各个部件或模块就有了运行的环境。

但是这个阶段的初始化远未完成，路还长得很，我们回到 start_kernel( )中再往下看。

[start_kernel( )]

```
532        printk("Kernel command line: %s\n", saved_command_line);
533        parse_options(command_line);
534        trap_init( );
535        init_IRQ( );
536        sched_init( );
537        time_init( );
538        softirq_init( );
539
540        /*
541         * HACK ALERT! This is early. We're enabling the console before
542         * we've done PCI setups etc, and console_init( ) must be aware of
543         * this. But we do want output early, in case something goes wrong.
```

```
544          */
545         console_init( );
```

这里调用的 init_IRQ( )、softirq_init( )以及 time_init( )都已在第 3 章中看过了，读者不妨回过去温习一下。我们要看的是其余几个函数。

首先是对引导命令行中各个选择项的分析和处理。前面在 setup_arch( )中(639 行)已经对命令行作过一些分析处理，但那只是专门针对与内存有关的选择项，现在则通过 parse_options( )对命令行进行全面的分析处理。函数 parse_options( )所作的只是一些字符串的处理，目的在于从命令行中分解出各个选择项，然后对每个选择项调用 checksetup( )加以处理。我们跳过 parse_options( )而直接看 checksetup( )，其代码在 init/main.c 中：

[start_kernel( ) > parse_options( ) > checksetup( )]

```
318     static int __init checksetup(char *line)
319     {
320         struct kernel_param *p;
321
322         p = &__setup_start;
323         do {
324             int n = strlen(p->str);
325             if (!strncmp(line, p->str, n)) {
326                 if (p->setup_func(line+n))
327                     return 1;
328             }
329             p++;
330         } while (p < &__setup_end);
331         return 0;
332     }
```

内核中有个数组，数组中的每个元素都是一个 kernel_param 数据结构，定义于 include/linux/init.h：

```
56     /*
57      * Used for kernel command line parameter setup
58      */
59     struct kernel_param {
60         const char *str;
61         int (*setup_func)(char *);
62     };
```

根据由 parse_options( )传下来的字符串，checksetup( )在这个数组中逐个搜索比对，如果与某一选择项的字符串相符，就执行由相应 kernel_param 数据结构通过函数指针提供的操作。那么，数组中的数据结构是在哪儿定义的呢？首先，在 include/linux/init.h 中定义了一个宏操作__setup( )：

```
66     #define __setup(str, fn)                          \
67         static char __setup_str_##fn[ ] __initdata = str;          \
68     static struct kernel_param __setup_##fn \
```

```
        __attribute__((unused)) __initsetup = {  __setup_str_##fn, fn }
69
70    #endif /* __ASSEMBLY__ */
```

这里的__initdata 和__initsetup 分别定义为：

```
78    #define __initdata __attribute__ ((__section__ (".data.init")))

80    #define __initsetup \__attribute__ ((unused, __section__  (".setup.init")))
```

表示在连接时要分别将有关的数据结构放在“.data.init”和“.setup.init”段中。

这样，以选择项“no387”为例，在 include/asm-i386/bugs.h 中有这么一行：

```
51    __setup("no387", no_387);
```

经过 gcc 的预处理以后，这一行就变成了这样：

```
static char __setup_str_no_387[ ] __initdata = "no387";                    \
static struct kernel_param __setup_no_387 __attribute__((unused)) __initsetup =
                                    { __setup_str_no_387,  no_387}
```

于是，如果命令行中有“no387”这个选择项，checksetup( )就会在字符串比对相符以后调用函数 no_387( )，其代码也在 include/asm-i386/bugs.h 中：

```
44    static int __init no_387(char *s)
45    {
46        boot_cpu_data.hard_math = 0;
47        write_cr0(0xE | read_cr0( ));
48        return 1;
49    }
```

对这个选择项的处理是很简单的，只是把 boot_cpu_data.hard_math 设成 0，并把控制寄存器%cr0 中的几个标志位设成 1，表示系统中没有 80387 协处理器，或者即使有也不用。

再如选择项“root=”，表示要将文件系统的总根建立在指定的设备上，其定义见 init/main.c：

```
316    __setup("root=", root_dev_setup);
```

相应的函数 root_dev_setup( )则为(init/main.c)：

```
299    static int __init root_dev_setup(char *line)
300    {
301        int i;
302        char ch;
303
304        ROOT_DEV = name_to_kdev_t(line);
```

```
305        memset (root_device_name, 0, sizeof root_device_name);
306        if (strncmp (line, "/dev/", 5) == 0) line += 5;
307        for (i = 0; i < sizeof root_device_name - 1; ++i)
308        {
309            ch = line[i];
310            if ( isspace (ch) || (ch == ',') || (ch == '\0') ) break;
311            root_device_name[i] = ch;
312        }
313        return 1;
314    }
```

显然，由 checksetup( )传下来的字符串已经跳过了前面的“root=”，而剩下的部分正是 root_dev_setup( )所关心的。代码本身很简单，我们就不加解释了。

这样的选择项有多少呢？有几十个之多。一般，如果内核中某个模块或设备驱动程序的设计人员觉得有需要提供一个引导选择项 xyz，就可以提供一个函数 do_xyz( )，并在程序中加上一行“__setup("xyz", do_xyz);”，这就行了。

下一个重要的函数是 trap_init( )，这个函数的作用是设置陷阱门和中断门，读者在第 3 章中已经看过它的代码。但是当时我们的注意力集中在中断机制，所以跳过了在 trap_init( )中调用的一个函数 cpu_init( )。

显然，cpu_init( )与 CPU 的初始化有关。如前所述，CPU 本身的初始化主要是在 startup_32( )中完成的，但是实际上有一部分层次比较高的初始化放在第二阶段才进行，这就是这里的 cpu_init( )。与 startup_32( )中的初始化相比，这里主要是为进程调度作准备，其代码在 arch/i386/kernel/setup.c 中：

[start_kernel( ) > trap_init( ) > cpu_init( )]

```
2199    /*
2200     * cpu_init( ) initializes state that is per CPU. Some data is already
2201     * initialized (naturally) in the bootstrap process, such as the GDT
2202     * and IDT. We reload them nevertheless, this function acts as a
2203     * 'CPU state barrier', nothing should get across.
2204     */
2205    void __init cpu_init (void)
2206    {
2207        int nr = smp_processor_id( );
2208        struct tss_struct * t = &init_tss[nr];
2209
2210        if (test_and_set_bit(nr, &cpu_initialized)) {
2211            printk("CPU#%d already initialized!\n", nr);
2212            for (;;) __sti( );
2213        }
2214        printk("Initializing CPU#%d\n", nr);
2215
2216        if (cpu_has_vme || cpu_has_tsc || cpu_has_de)
2217            clear_in_cr4(X86_CR4_VME|X86_CR4_PVI|X86_CR4_TSD|X86_CR4_DE);
2218    #ifndef CONFIG_X86_TSC
```

```
2219        if (tsc_disable && cpu has tsc) {
2220            printk("Disabling TSC...\n");
2221            /**** FIX-HPA: DOES THIS REALLY BELONG HERE? ****/
2222            clear_bit(X86_FEATURE_TSC, boot_cpu_data.x86_capability);
2223            set_in_cr4(X86_CR4_TSD);
2224        }
2225    #endif
2226
2227        __asm__ __volatile__("lgdt %0": "=m" (gdt_descr));
2228        __asm__ __volatile__("lidt %0": "=m" (idt descr));
2229
2230        /*
2231         * Delete NT
2232         */
2233        __asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");
2234
2235        /*
2236         * set up and load the per-CPU TSS and LDT
2237         */
2238        atomic_inc(&init_mm.mm_count);
2239        current->active mm = &init_mm;
2240        if(current->mm)
2241            BUG( );
2242        enter_lazy_tlb(&init mm, current, nr);
2243
2244        t->esp0 = current->thread.esp0;
2245        set tss_desc(nr,t);
2246        gdt table[__TSS(nr)].b &= 0xfffffdff;
2247        load TR(nr);
2248        load_LDT(&init mm);
2249
2250        /*
2251         * Clear all 6 debug registers:
2252         */
2253
2254    #define CD(register) __asm__("movl %0,%%db" #register ::"r"(0) );
2255
2256        CD(0); CD(1); CD(2); CD(3); /* no db4 and db5 */; CD(6); CD(7);
2257
2258    #undef CD
2259
2260        /*
2261         * Force FPU initialization:
2262         */
2263        current->flags &= ~PF_USEDFPU;
2264        current->used_math = 0;
2265        stts( );
2266    }
```

这个函数不仅由主 CPU 在 trap_init( )中调用，也由次 CPU 在进入 start_secondary( )以后调用。代码中的 cpu_initialized 是个全局的位图，系统中的每个 CPU 在这个位图中都有个对应的标志位，一个 CPU 完成了本身的初始化就将相应的标志位设置成 1，让其他 CPU，其实主要是主 CPU，知道这个 CPU 的初始化已经完成了。对于每个具体的 CPU 而言，将 cpu_initialized 中的对应位设置成 1 只能发生一次。如果有哪一个 CPU 再次试图设置其对应标志位就肯定是出了问题，所以就让出了问题的 CPU 陷入一个无限循环。这实际上相当于让 CPU 跳过下面的所有操作进入空转，因为 CPU 在这个循环中仍能响应中断请求，并且在中断服务以后若发现需要调度就会如常进行调度。不过，CPU 在所谓“空转”中会进入停机状态，实际上是不转，而在这个 for 循环中则真正在打转。

跳过对 TSC 等的设置，下面对段寄存器的设置、对(当前进程的)task_struct 结构中的指针 active_mm 的设置，以及为什么 task_struct 结构中的指针 mm 应该是 0，还有对“任务寄存器”的设置，结合第 3 章和第 4 章的内容读者应该能自己读懂。代码中的 2233 行将 CPU 中“标志位寄存器”的 NT 位(表示嵌套进入内核，“Nested Task”)清成 0。由于没有直接对这个寄存器进行位操作的指令，这里先把它压入堆栈，在堆栈中操作完了再送回寄存器。2248 行通过 load_LDT( )装入局部段描述表指针 LDTR，不过 Linux 内核只在 VM86 模式下才使用局部段，所以我们在这里并不关心。还有，2256 行是对 CPU 中一些程序调试寄存器的初始化，2263~2265 行是对浮点处理器的初始化，这里就都从略了。

再看对进程调度机制的初始化。

[start_kernel( ) > sched_init( )]

```
1244    void __init sched_init(void)
1245    {
1246        /*
1247         * We have to do a little magic to get the first
1248         * process right in SMP mode.
1249         */
1250        int cpu = smp_processor_id( );
1251        int nr;
1252
1253        init_task.processor = cpu;
1254
1255        for(nr = 0; nr < PIDHASH_SZ; nr++)
1256            pidhash[nr] = NULL;
1257
1258        init_timervecs( );
1259
1260        init_bh(TIMER_BH, timer_bh);
1261        init_bh(TQUEUE_BH, tqueue_bh);
1262        init_bh(IMMEDIATE_BH, immediate_bh);
1263
1264        /*
1265         * The boot idle thread does lazy MMU switching as well:
1266         */
1267        atomic_inc(&init_mm.mm_count);
1268        enter_lazy_tlb(&init_mm, current, cpu);
1269    }
```

这里所调用的一些函数对读者已经不陌生了。虽然还没有看过 init_timervecs( )的代码，但是顾名思义这是对定时器中断向量的初始化。所以，我们把这个函数留给读者。

然后是对系统主控终端的初始化，读者已经在“设备驱动”一章中看到过了。

我们还得往下看。

[start_kernel( )]

```
546    #ifdef CONFIG_MODULES
547        init_modules( );
548    #endif
549        if (prof_shift) {
550            unsigned int size;
551            /* only text is profiled */
552            prof_len = (unsigned long) & etext - (unsigned long) & stext;
553            prof_len >>= prof_shift;
554
555            size = prof_len * sizeof(unsigned int) + PAGE_SIZE-1;
556            prof_buffer = (unsigned int *) alloc_bootmem(size);
557        }
558
559        kmem_cache_init( );
560        sti( );
561        calibrate_delay( );
562    #ifdef CONFIG_BLK_DEV_INITRD
       ......
569    #endif
570        mem_init( );
571        kmem_cache_sizes_init( );
572    #ifdef CONFIG_3215_CONSOLE
573            con3215_activate( );
574    #endif
575    #ifdef CONFIG_PROC_FS
576        proc_root_init( );
577    #endif
578        mempages = num_physpages;
579
580        fork_init(mempages);
581        proc_caches_init( );
582        vfs_caches_init(mempages);
583        buffer_init(mempages);
584        page_cache_init(mempages);
585        kiobuf_setup( );
586        signals_init( );
587        bdev_init( );
588        inode_init(mempages);
589    #if defined(CONFIG_SYSVIPC)
590        ipc_init( );
```

```
591     #endif
592     #if defined(CONFIG_QUOTA)
593         dquot_init_hash( );
594     #endif
595         check_bugs( );
596         printk("POSIX conformance testing by UNIFIX\n");
```

首先是对可安装模块这个机制的初始化，其实只是计算出内核符号表的大小，而且计算也很简单。函数 init_modules( )的代码在 kernel/module.c 中：

[start_kernel( ) > init_modules( )]

```
229     /*
230      * Called at boot time
231      */
232
233     void __init init_modules(void)
234     {
235         kernel_module.nsyms = __stop___ksymtab - __start___ksymtab;
236
237     #ifdef __alpha__
        ......
239     #endif
240     }
```

物理页面 empty_zero_page 在前面被借用来传递引导命令行和参数块，现在应该物归原主了。顾名思义，这个页面的内容应该是全 0。此外，基本内存中的页面，即 1MB 以下的页面还没有释放以供动态分配，前面使用的页面位图所在的页面现在也已经完成了使命，也应该释放以供动态分配。还有，前面讲过，在页面映射目录中的低区，即对应于用户空间的区间有几个临时性的目录项，最后应该予以清除。同时，还需要收集或计算出一些统计信息。这些操作都是由 mem_init( )完成的，其代码在 mm/init.c 中：

[start_kernel( ) > mem_init( )]

```
554     void __init mem_init(void)
555     {
556         int codesize, reservedpages, datasize, initsize;
557         int tmp;
558
559         if (!mem_map)
560             BUG( );
561
562     #ifdef CONFIG_HIGHMEM
563         highmem_start_page = mem_map + highstart_pfn;
564         max_mapnr = num_physpages = highend_pfn;
565     #else
```

```
566         max mapnr = num_physpages = max_low_pfn;
567     #endif
568         high_memory = (void *) __va(max_low_pfn * PAGE_SIZE);
569 
570         /* clear the zero-page */
571         memset(empty_zero_page, 0, PAGE_SIZE);
572 
573         /* this will put all low memory onto the freelists */
574         totalram_pages += free_all_bootmem( );
575 
576         reservedpages = 0;
577         for (tmp = 0; tmp < max_low_pfn; tmp++)
578             /*
579              * Only count reserved RAM pages
580              */
581             if (page_is_ram(tmp) && PageReserved(mem_map+tmp))
582                 reservedpages++;
583     #ifdef CONFIG_HIGHMEM
584         for (tmp = highstart_pfn; tmp < highend_pfn; tmp++) {
585             struct page *page = mem_map + tmp;
586 
587             if (!page_is_ram(tmp)) {
588                 SetPageReserved(page);
589                 continue;
590             }
591             ClearPageReserved(page);
592             set_bit(PG_highmem, &page->flags);
593             atomic_set(&page->count, 1);
594             __free_page(page);
595             totalhigh_pages++;
596         }
597         totalram_pages += totalhigh_pages;
598     #endif
599         codesize =  (unsigned long) &_etext - (unsigned long) &_text;
600         datasize =  (unsigned long) &_edata - (unsigned long) &_etext;
601         initsize =  (unsigned long) &__init_end - (unsigned long) &__init_begin;
602 
603       printk("Memory: %luk/%luk available \
          (%dk kernel code, %dk reserved, %dk data, %dk init, %ldk highmem)\n",
604             (unsigned long) nr_free_pages( ) << (PAGE_SHIFT-10),
605             max_mapnr << (PAGE_SHIFT-10),
606             codesize >> 10,
607             reservedpages << (PAGE_SHIFT-10),
608             datasize >> 10,
609             initsize >> 10,
610             (unsigned long) (totalhigh_pages << (PAGE_SHIFT-10))
611                );
612 
```

```
613    #if CONFIG_X86_PAE
614        if (!cpu_has_pae)
615            panic("cannot execute a PAE-enabled kernel on a PAE-less CPU!");
616    #endif
617        if (boot_cpu_data.wp_works_ok < 0)
618            test_wp_bit( );
619
620        /*
621         * Subtle. SMP is doing it's boot stuff late (because it has to
622         * fork idle threads) - but it also needs low mappings for the
623         * protected-mode entry to work. We zap these entries only after
624         * the WP-bit has been tested.
625         */
626    #ifndef CONFIG_SMP
627        zap_low_mappings( );
628    #endif
629
630    }
```

这里的 571 行将 empty_zero_page 的内容清成全 0。接着调用的 free_all_bootmem( )是个重要的操作，它根据页面位图的内容释放物理内存中所有可供动态分配的页面，将这些页面投入动态分配。

[start_kernel( ) > mem_init( ) > free_all_bootmem( )]

```
300    unsigned long __init free_all_bootmem (void)
301    {
302        return(free_all_bootmem_core(&contig_page_data));
303    }
```

[start_kernel( ) > mem_init( ) > free_all_bootmem( ) > free_all_bootmem_core( )]

```
224    static unsigned long __init free_all_bootmem_core(pg_data_t *pgdat)
225    {
226        struct page *page = pgdat->node_mem_map;
227        bootmem_data_t *bdata = pgdat->bdata;
228        unsigned long i, count, total = 0;
229        unsigned long idx;
230
231        if (!bdata->node_bootmem_map) BUG( );
232
233        count = 0;
234        idx = bdata->node_low_pfn - (bdata->node_boot_start >> PAGE_SHIFT);
235        for (i = 0; i < idx; i++, page++) {
236            if (!test_bit(i, bdata->node_bootmem_map)) {
237                count++;
238                ClearPageReserved(page);
239                set_page_count(page, 1);
240                __free_page(page);
```

```
241             }
242         }
243         total += count;
244 
245         /*
246          * Now free the allocator bitmap itself, it's not
247          * needed anymore:
248          */
249         page = virt_to_page(bdata->node_bootmem_map);
250         count = 0;
251       for (i = 0;
              i < ((bdata->node_low_pfn-(bdata->node_boot_start >> PAGE_SHIFT))/8
                   + PAGE_SIZE-1)/PAGE_SIZE;
              i++, page++) {
252             count++;
253             ClearPageReserved(page);
254             set_page_count(page, 1);
255             __free_page(page);
256         }
257         total += count;
258         bdata->node_bootmem_map = NULL;
259 
260         return total;
261     }
```

这里的 bdata->node_boot_start 是在前面的 init_bootmem_core( )中设置的，表示整个物理内存的起点(页面号)，其数值为 0，但是在特殊的情况下也可以不是 0。而 bdata->node_low_pfn 则为物理内存中最高的页面号（不包括 HIGHMEM）。235～242 行的 for 循环扫描整个物理内存的所有页面，对于每个页面都测试页面位图中的对应位，如果对应位为 0 就表示这个页面可以投入分配。把相应 page 结构中的使用计数设置成 1，假装这个页面已经分配，再对其调用__free_page( )，就把这个 page 结构挂入了所属的空闲页面队列。可以说，前面为物理页面管理机制和空闲页面队列的建立作了很多的准备，特别是页面位图的设立，最终为的就是这一步。正因为这样，一旦完成了这一步，页面位图就不再需要，因而它所占据的物理页面也可以释放了，251～256 行的 for 循环就是用来释放这些页面。

在 mem_init( )中还有条件地调用了一个函数 test_wp_bit( )，这是针对 80386 CPU 而设的，现在已经没有多大意义了。注意 627 行对 zap_low_mappings( )的调用是条件编译的语句，仅对单处理器系统有效。在初始化的第一阶段建立起了页面映射目录，为了向页式映射过渡，当时的目录中同时在系统空间和用户空间映射了物理内存的开头 8MB 空间。后来把系统空间的映射扩展到了整个物理内存，但是其用户空间的映射尚未拆除。而内核线程在正常运行中是不应该使用 0xC0000000 以下的地址的，所以应该把用户空间的映射拆除。然而，在 SMP 结构的系统中，现在次 CPU 尚未开始运行，当次 CPU 开始运行时也要经历初始化的第一阶段，也要向页式映射过渡，所以现在还不能拆除这些映射。

回到 start_kernel( )的代码中，kmem_cache_sizes_init( )是对内核中通用 slab 缓冲区管理机制的初始化，通过 kmem_cache_create( )建立起大小分别为 32 字节、64 字节、…、128KB 的缓冲区队列，这些 slab 缓冲区是供内核通过 kmalloc( )动态分配的，专用的 slab 缓冲区则不在其中。接着，proc_root_init( )是对特殊文件系统/proc 的初始化，读者已经在第 5 章看过这个函数的代码。再下面是 fork_init( )，根

据物理内存的大小计算出允许创建线程(包括进程)的数量，其代码在 kernel/fork.c 中：

[start_kernel( ) > fork_init( )]

```
77    void __init fork_init(unsigned long mempages)
78    {
79    /*
80    * The default maximum number of threads is set to a safe
81    * value: the thread structures can take up at most half
82    * of memory.
83    */
84    max_threads = mempages / (THREAD_SIZE/PAGE_SIZE) / 2;
85
86    init_task.rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;
87    init_task.rlim[RLIMIT_NPROC].rlim_max = max_threads/2;
88    }
```

接下来的 proc_caches_init( )、vfs_caches_init( )、buffer_init( )以及 kiobuf_setup( )和 signals_init( )，还有 bdev_init( ) 和 inode_init( )，基本上都是为有关的管理机制建立起专用 slab 缓冲区队列。而 page_cache_init( )则分配空间建立起缓冲页面杂凑表 page_hash_table，这个函数的代码在 mm/filemap.c 中：

[start_kernel( ) > page_cache_init( )]

```
2583    void __init page_cache_init(unsigned long mempages)
2584    {
2585        unsigned long htable_size, order;
2586
2587        htable_size = mempages;
2588        htable_size *= sizeof(struct page *);
2589        for(order = 0; (PAGE_SIZE << order) < htable_size; order++)
2590            ;
2591
2592        do {
2593            unsigned long tmp = (PAGE_SIZE << order) / sizeof(struct page *);
2594
2595            page_hash_bits = 0;
2596            while((tmp >>= 1UL) != 0UL)
2597                page_hash_bits++;
2598
2599            page_hash_table = (struct page **)
2600                __get_free_pages(GFP_ATOMIC, order);
2601        } while(page_hash_table == NULL && --order > 0);
2602
2603        printk("Page-cache hash table entries: %d (order: %ld, %ld bytes)\n",
2604              (1 << page_hash_bits), order, (PAGE_SIZE << order));
2605        if (!page_hash_table)
```

```
2606            panic("Failed to allocate page hash table\n");
2607        memset((void *)page_hash_table, 0, PAGE_HASH_SIZE * sizeof(struct page *));
2608    }
```

这里引用的宏操作定义于 include/linux/pagemap.h：

```
42    #define PAGE_HASH_BITS (page_hash_bits)
43    #define PAGE_HASH_SIZE (1 << PAGE_HASH_BITS)
```

缓冲页面杂凑表是个 page 结构指针数组，数组中的每一项都用来维持一个 page 结构的单链队列。这个杂凑表的使用对于提高文件操作和页面换入/换出的效率起着重要的作用。杂凑表的大小是 2 的整数次幂，大的杂凑表有利于使页面在队列中更趋分散，从而有利于提高查找效率，但是太大了也没有必要。所以，这里根据物理内存的大小先估算出一个期望值，然后试着按这个大小分配空间，如果分配不到就降格以求，减小对大小的要求。最后，成功分配到所需的空间以后，就把它初始化成全 0。

至于 ipc_init( )，那当然是对 Sys V 进程间通信机制的初始化。

[start_kernel( ) > ipc_init( )]

```
34    void __init ipc_init (void)
35    {
36        sem_init( );
37        msg_init( );
38        shm_init( );
39        return;
40    }
```

这些函数大体上都一样，都是对有关数据结构如 msg_ids、shm_ids 的初始化，同时为通过/proc 特殊文件系统获取有关信息创造条件，在/proc 目录下建立起“sysvipc/shm”等节点。

初始化的第二阶段终于接近尾声了，下面是最后的冲刺。

[start_kernel( )]

```
597
598        /*
599         *  We count on the initial thread going ok
600         *  Like idlers init is an unlocked kernel thread, which will
601         *  make syscalls (and thus be locked).
602         */
603        smp_init( );
604        kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
605        unlock_kernel( );
606        current->need_resched = 1;
607        cpu_idle( );
608    }
```

迄今为止的初始化一直都只有一个 CPU 在执行(尽管有些代码是公用的)。如果是 SMP 结构的系统，

就一直只有主 CPU 在运行，其余的 CPU 都是“次 CPU”，都要受到主 CPU 的启动才能运行。现在，系统的“经济基础”已经建立，可以启动这些次 CPU 的运行了。这里的 smp_init( )就是对 SMP 结构的初始化，具体包括启动系统中的各个次 CPU，让它们完成自身的初始化并进入各自的空转进程。读者已经在前一章中看到这个函数的代码以及对有关过程的介绍。

从 smp_init( )返回到 start_kernel( )时，SMP 结构的初始化已经完成，所有的次 CPU 都已经或者即将经由 start_secondary( )进入 cpu_idle( )。主 CPU 在初始化的第二阶段还有一件事要做，那就是创建内核线程 init( )。创建了这个内核线程以后，主 CPU 也会转入 cpu_idle( )。可见，不管是主 CPU 还是次 CPU，最终都会进入 cpu_idle( )。当系统中所有的 CPU 都进入 cpu_idle( )中时，它们就站在了同一个起点上，开始公平竞争了。这个函数的代码在 kernel/process.c 中：

[start_kernel( ) > cpu_idle( )]

```
117    /*
118     * The idle thread. There's no useful work to be
119     * done, so just try to conserve power and have a
120     * low exit latency (ie sit in a loop waiting for
121     * somebody to say that they'd like to reschedule)
122     */
123    void cpu_idle (void)
124    {
125        /* endless idle loop with no priority at all */
126        init_idle( );
127        current->nice = 20;
128        current->counter = -100;
129
130        while (1) {
131            void (*idle)(void) = pm_idle;
132            if (!idle)
133                idle = default_idle;
134            while (!current->need_resched)
135                idle( );
136            schedule( );
137            check_pgt_cache( );
138        }
139    }
```

这个函数的主体是个无穷循环，一进入这个循环就再也出不来了。既然所有的 CPU 都在执行这个循环，而且在循环中也并不引用与此前的历史有关的变量，也就分不出什么主次，所以从此以后系统中所有的 CPU 就都互相平等了。当一个 CPU 在 cpu_idle( )中执行时，就是处于该 CPU 的“空转”进程中。读者在前面已经看到，空转进程并不挂入系统中的就绪队列，其 task_struct 结构指针保持在以 CPU 编号为下标的数组 init_tasks[ ]中。由于空转进程不挂入就绪队列，在进程调度时就永远不会以空转进程为目标。相反，只有在没有进程可以运行时才会回到空转进程中来。

那么，CPU 在空转进程中干些什么呢？简言之就是循环地执行一个空转函数，直到进程的 task_struct 结构中的 need_resched 变成 1，此时就通过 schedule( )进行一次进程调度。CPU 一进入

schedule( )，就一直要到系统中再无就绪进程可以运行时才会返回到 cpu_idle( )中，然后就再等待 current->need_resched 变成 1。一般情况下的空转函数为 default_idle( )，其代码见 kernel/process.c（在引导时可以在命令行中通过选择项“idle=poll”将 pm_idle 设置成指向 poll_idle( )，但是两个函数并无本质的不同）。

[cpu_idle( ) > default_idle( )]

```
80      static void default_idle(void)
81      {
82          if (current_cpu_data.hlt_works_ok && !hlt_counter) {
83              __cli( );
84              if (!current->need_resched)
85                  safe_halt( );
86              else
87                  __sti( );
88          }
89      }

300     /* used in the idle loop; sti takes one instruction cycle to complete */
301     #define safe_halt( )        __asm__ __volatile__("sti; hlt": : :"memory")
```

当CPU在cpu_idle( )中，而task_struct结构中的need_resched又为 0 时，就是实在无事可干了，所以把中断打开，并通过“hlt”指令进入“停机” 状态，或者说硬件睡眠状态。处于这个状态的CPU不执行任何指令，直到有中断请求到来时才会恢复运行，那时就好像刚结束hlt指令的执行一样。然后，当CPU结束对中断的处理时，就会回到hlt指令的后继指令，那就是从default_idle( )返回了。取决于具体的中断，在中断服务程序中有可能会使当前进程的need_resched标志变成 1，那样就会在cpu_idle( )中结束 134 行的while循环而调用schedule( )，否则就又进入default_idle( )。

当系统中所有的CPU都进入cpu_idle( )中时，就绪队列中只有一个进程，那就是前面 604 行创建的 init( )。所以，实际上只有一个CPU能在schedule( )中调度这个惟一的进程运行，因此只让一个CPU执行 schedule( )就行了。让谁执行schedule( )呢？“近水楼台先得月”，原先的主CPU在放下身份与次CPU“打成一片”之前留了一手，在 606 行把它自己的need_resched标志设成了 1，从而成了最先进行调度的CPU，当仁不让地取得了执行init( )的权利。而其他的CPU，则自会在cpu_idle( )中的 134 行受到阻挡而暂时不能前进。

10.4　系统初始化（第三阶段）

内核线程 init( )的任务仍然还是初始化，当然是进一步的、更高层次上的初始化。事实上，从引导结束，CPU 转入内核映象开始，一共有三个阶段的初始化。第一个阶段是从进入 startup_32( )开始到进入 start_kernel( )或者 start_secondary( )。这个阶段主要是对 CPU 自身的初始化，主 CPU 和次 CPU 都要经历这种初始化，但是主 CPU 要多作一些贡献。第二个阶段是从进入 start_kernel( )开始到进入 cpu_idle( )。这个阶段主要是对系统的“经济基础”，即各种资源的初始化，仅由主 CPU 进行。第三个阶段则是 init( )的执行，这是对系统的“上层建筑”的初始化。表面上此时已无主 CPU 和次 CPU 之分，

由谁执行 init( )取决于竞争调度的结果，但是由于主 CPU 预先留了一手，实际上总是由它执行的。函数 init( )的代码在 init/main.c 中，这个函数本身并不长，但是实际进行的操作却不简单，所以我们还是要分段阅读。

```
761	static int init(void * unused)
762	{
763		lock_kernel( );
764		do_basic_setup( );
765	
766		/*
767		 * Ok, we have completed the initial bootup, and
768		 * we're essentially up and running. Get rid of the
769		 * initmem segments and start the user-mode stuff..
770		 */
771		free_initmem( );
772		unlock_kernel( );
773	
```

这里在对内核加锁的条件下调用了 do_basic_setup( )和 free_initmem( )两个函数。表面上此时系统中只有一个进程可以被调度运行，因而只有一个 CPU 能够运行，似乎不必加锁。但是，下面就会看到，在执行 init( )的过程中还会创建新的进程。另一方面，如果发生中断，则这些 CPU 还是有可能在中断服务程序中造成干扰。

先看 do_basic_setup( )，这是主要的，其代码在 init/main.c 中。我们删去了代码中一些条件编译的片段，另一些条件编译的片段因为本来就只有一行语句而没有删去，但是我们在这里基本上不关心这些选择项，有兴趣或需要的读者可自行研读。这些(我们不关心的)条件编译选择项有：

- CONFIG_BLK_DEV_INITRD。用于 RAMDISK，即以一部分内存来模拟硬盘。
- CONFIG_MTRR。如果 CPU 中有 MTRR 寄存器，就可以通过这个寄存器按区间来管理内存的高速缓冲。不过，也可以不用 MTRR，而按页面来管理高速缓冲。
- CONFIG_SYSCTL。允许在运行中动态地改变一些内核中的参数。
- CONFIG_SBUS。SBUS 是 Sparc 工作站中采用的总线。
- CONFIG_PPC。适用于 Power PC 处理器。
- CONFIG_MCA。用于 PS/2 系统结构中的 Micro Channel 外设总线。
- CONFIG_ARCH_ACORN。适用于 ARM 处理器。
- CONFIG_ZORRO。一种总线。
- CONFIG_DIO。一种总线。
- CONFIG_NUBUS。Macintosh 计算机中的总线。
- CONFIG_ISAPNP。用于 ISA 总线上支持 PnP（Plug and Play）的接口卡。
- CONFIG_TC。用于一种称为 Turbo Channel 的总线。
- CONFIG_IRDA。用于红外线通信接口。
- CONFIG_PCMCIA。用于笔记本电脑的外插接口卡。

至于 CONFIG_PCI，则虽然也是选择项(笔记本电脑中没有 PCI 总线)，但是实际上不但用于桌上型 PC，也广泛用于嵌入式系统中，读者已经在“设备驱动”一章中看过其初始化函数 pci_init( )的代码。

[init( ) > do_basic_setup( )]

```
641     /*
642      * Ok, the machine is now initialized. None of the devices
643      * have been touched yet, but the CPU subsystem is up and
644      * running, and memory and process management works.
645      *
646      * Now we can finally start doing some real work..
647      */
648     static void __init do_basic_setup(void)
649     {
650     #ifdef CONFIG_BLK_DEV_INITRD
        ......
652     #endif
653
654         /*
655          * Tell the world that we're going to be the grim
656          * reaper of innocent orphaned children.
657          *
658          * We don't want people to have to make incorrect
659          * assumptions about where in the task array this
660          * can be found.
661          */
662         child_reaper = current;
663
664     #if defined(CONFIG_MTRR)    /* Do this after SMP initialization */
665     /*
666      * We should probably create some architecture-dependent "fixup after
667      * everything is up" style function where this would belong better
668      * than in init/main.c..
669      */
670         mtrr_init( );
671     #endif
672
673     #ifdef CONFIG_SYSCTL
674         sysctl_init( );
675     #endif
676
677         /*
678          * Ok, at this point all CPU's should be initialized, so
679          * we can start looking into devices..
680          */
681     #ifdef CONFIG_PCI
682         pci_init( );
683     #endif
684     #ifdef CONFIG_SBUS
685         sbus_init( );
686     #endif
```

```
687   #if defined(CONFIG_PPC)
688       ppc_init( );
689   #endif
690   #ifdef CONFIG_MCA
691       mca_init( );
692   #endif
693   #ifdef CONFIG_ARCH_ACORN
694       ecard_init( );
695   #endif
696   #ifdef CONFIG_ZORRO
697       zorro_init( );
698   #endif
699   #ifdef CONFIG_DIO
700       dio_init( );
701   #endif
702   #ifdef CONFIG_NUBUS
703       nubus_init( );
704   #endif
705   #ifdef CONFIG_ISAPNP
706       isapnp_init( );
707   #endif
708   #ifdef CONFIG_TC
709       tc_init( );
710   #endif
711
712       /* Networking initialization needs a process context */
713       sock_init( );
714
715   #ifdef CONFIG_BLK_DEV_INITRD
      ......
720   #endif
721
722       start_context_thread( );
723       do_initcalls( );
724
725       /* .. filesystems .. */
726       filesystem_setup( );
727
728   #ifdef CONFIG_IRDA
729       irda_device_init( ); /* Must be done after protocol initialization */
730   #endif
731   #ifdef CONFIG_PCMCIA
732       init_pcmcia_ds( );       /* Do this last */
733   #endif
734
735       /* Mount the root filesystem.. */
736       mount_root( );
737
```

```
738         mount_devfs_fs ( );
739
740     #ifdef CONFIG_BLK_DEV_INITRD
        . . . . . . .
758     #endif
759     }
```

跳过条件编译部分，剩下也就不多了。我们把 sock_init( )的代码留给读者结合第 7 章自己阅读，这里先看 start_context_thread( )的代码，它在 kernel/context.c 中：

[init( ) > do_basic_setup( ) > start_context_thread( )]

```
149     int start_context_thread(void)
150     {
151       kernel_thread(context_thread, NULL,
                                        CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
152         return 0;
153     }
```

这是系统中创建的第二个内核线程，是一个所谓“守护神”(daemon)，或者说“代理人”，名叫“keventd”。其代码 context_thread( )也在 kernel/context.c 中：

```
66      static int context_thread(void *dummy)
67      {
68          struct task_struct *curtask = current;
69          DECLARE_WAITQUEUE(wait, curtask);
70          struct k_sigaction sa;
71
72          daemonize( );
73          strcpy(curtask->comm, "keventd");
74          keventd_running = 1;
75          keventd_task = curtask;
76
77          spin_lock_irq(&curtask->sigmask_lock);
78          siginitsetinv(&curtask->blocked, sigmask(SIGCHLD));
79          recalc_sigpending(curtask);
80          spin_unlock_irq(&curtask->sigmask_lock);
81
82          /* Install a handler so SIGCLD is delivered */
83          sa.sa.sa_handler = SIG_IGN;
84          sa.sa.sa_flags = 0;
85          siginitset(&sa.sa.sa_mask, sigmask(SIGCHLD));
86          do_sigaction(SIGCHLD, &sa, (struct k_sigaction *)0);
87
88          /*
89           * If one of the functions on a task queue re-adds itself
90           * to the task queue we call schedule( ) in state TASK_RUNNING
```

```
91          */
92          for (;;) {
93              set_task_state(curtask, TASK_INTERRUPTIBLE);
94              add_wait_queue(&context_task_wq, &wait);
95              if (TQ_ACTIVE(tq_context))
96                  set_task_state(curtask, TASK_RUNNING);
97              schedule( );
98              remove_wait_queue(&context_task_wq, &wait);
99              run_task_queue(&tq_context);
100             wake_up(&context_task_done);
101             if (signal_pending(curtask)) {
102                 while (waitpid(-1, (unsigned int *)0, __WALL|WNOHANG) > 0)
103                     ;
104                 flush_signals(curtask);
105                 recalc_sigpending(curtask);
106             }
107         }
108     }
```

这个进程的主体是一个无穷 for 循环，平时在一个队列 context_task_wq 中睡眠等待。在一些设备驱动程序中，有一些函数可能需要在一个进程的上下文中执行(而不是在中断处理的上下文中执行)，keventd 就为这种函数充当着“代理人”的角色。需要把一个函数提交给 keventd，使这个函数得以在 keventd 的上下文中执行时，就为这个函数准备下一个 tq_struct 数据结构，然后通过 schedule_task( )把这个数据结构挂入队列 tq_context，并唤醒在队列 context_task_wq 中睡眠的 keventd。当 keventd 被调度运行时，就会以它的名义、在它的上下文中通过 run_task_queue( )执行挂在 tq_context 中的函数。函数 schedule_task( )的代码在 kernel/context.c 中，读者可与 context_thread( )的代码对照着阅读。

```
44      /**
45       * schedule_task - schedule a function for subsequent execution in process context.
46       * @task: pointer to a &tq_struct which defines the function to be scheduled.
47       *
48       * May be called from interrupt context. The scheduled function is run at some
49       * time in the near future by the keventd kernel thread.  If it can sleep, it
50       * should be designed to do so for the minimum possible time, as it will be
51       * stalling all other scheduled tasks.
52       *
53       * schedule_task( ) returns non-zero if the task was successfully scheduled.
54       * If @task is already residing on a task queue then schedule_task( ) fails
55       * to schedule your task and returns zero.
56       */
57      int schedule_task(struct tq_struct *task)
58      {
59          int ret;
60          need_keventd(__FUNCTION__);
61          ret = queue_task(task, &tq_context);
```

```
62          wake_up(&context_task_wq);
63          return ret;
64      }
```

那么，这种函数与第 3 章中的 bh 函数有什么区别呢？我们讲过，bh 函数实质上是中断服务程序的一部分，是在中断服务的上下文中执行的，因而受到一些限制，而在进程上下文中执行的函数就不受这些限制。例如，在 bh 函数中就不应该有可能受阻而需要睡眠等待的操作，而在进程上下文中执行的函数则可以使当前进程，即 keventd 线程进入睡眠。

从 context_thread( )的代码中还可以看出，一般的内核线程要调用一个函数 daemonize( )才能变成“守护神”，这个函数的作用其实只是释放从父进程继承下来的一些资源(斩断尘缘才能成神)，而改换门庭投到 init_task 的门下，共享它的资源。

[context_thread( ) > daemonize( )]

```
1197    /*
1198     * Put all the gunge required to become a kernel thread without
1199     * attached user resources in one place where it belongs.
1200     */
1201
1202    void daemonize(void)
1203    {
1204        struct fs_struct *fs;
1205
1206
1207        /*
1208         * If we were started as result of loading a module, close all of the
1209         * user space pages.  We don't need them, and if we didn't close them
1210         * they would be locked into memory.
1211         */
1212        exit_mm(current);
1213
1214        current->session = 1;
1215        current->pgrp = 1;
1216
1217        /* Become as one with the init task */
1218
1219        exit_fs(current);    /* current->fs->count--; */
1220        fs = init_task.fs;
1221        current->fs = fs;
1222        atomic_inc(&fs->count);
1223        exit_files(current);
1224        current->files = init_task.files;
1225        atomic_inc(&current->files->count);
1226    }
```

回到 do_basic_setup( )，调用的下一个函数是 do_initcalls( )，其代码在 init/main.c 中：

[init( ) > do_basic_setup( ) > do_initcalls( )]

```
627    static void __init do_initcalls(void)
628    {
629        initcall_t *call;
630
631        call = &__initcall_start;
632        do {
633            (*call)( );
634            call++;
635        } while (call < &__initcall_end);
636
637        /* Make sure there is no pending stuff from the initcall sequence */
638        flush_scheduled_tasks( );
639    }
```

这个函数貌不惊人，可是干的事情却非同小可！

大家都知道，内核映象也跟一般可执行程序的映象一样有代码段和数据段。但是这只是一般而言，事实上内核映象中还有其他一些比较小的段和子段。内核源代码中有个文件 arch/i386/vmlinux.lds 是对连接工具 GNU ld 的连接说明，文件中描述了内核映象中所有的段和子段，读者不妨看一下。我们在这里关心的是其中的两个，一个是“.text.init”、我们称之为“初始化代码段”，另一个则是“.initcall.init”、我们称之为“初始化调用段”。有关的定义如下(linux/init.h)：

```
43    /*
44     * Used for initialization calls..
45     */
46    typedef int (*initcall_t)(void);
47    typedef void (*exitcall_t)(void);
48
49    extern initcall_t __initcall_start, __initcall_end;
      ......
51    #define __initcall(fn)                                    \
52        static initcall_t __initcall_##fn __init_call = fn
      ......
76    #define __init      __attribute__ ((__section__ (".text.init")))
      ......
81    #define __init_call \
                  __attribute__ ((unused, __section__ (".initcall.init")))
```

内核的代码中，凡是只在系统初始化时调用一次，以后就不会再受到调用的函数就在前面加上一个限定词__init，例如上面的 do_initcalls( )本身就是这样的一个函数。这些函数在编译、连接以后就全都集中在一起，都在“初始化代码段”中。之所以要这样，是因为这些函数所占的内存空间在初始化完成以后就可以另作他用，而集中在一起就可以成片地回收。

“初始化代码段”中的函数不是一律平等的，其中有些函数是某一方面初始化的“入口函数”，所以需要在系统初始化时从 do_initcalls( )中加以调用，而同在初始化代码段中的另一些函数则自会因此

而辗转地受到调用。对“入口函数”的调用通常都是独立的，并不互相依赖，所以孰先孰后没有什么关系。可是怎样调用呢？当然可以按常规把对这些函数的调用全都列出在 do_initcalls( )中。但是这样多少会给内核代码的编写和维护带来一些不便，例如每要增加或减少一种什么功能时就得相应地修改 do_initcalls( )的代码。为了减少这种不便，内核代码的作者们采用了一个巧妙的办法，那就是为这些需要在初始化时加以调用的函数都准备下一个函数指针，并把这些函数指针都集中在一起形成一个“初始化调用段”，实际上就是一个函数指针数组。这样，就可以通过一个循环来调用所有这些函数了，这就是 do_initcalls( )中的 do-while 循环的来历，代码中的__initcall_start 和__initcall_end 分别是这个段的起点和终点。这样的安排为编程带来了方便。例如，假定你要在内核中增加一个功能模块 xyz，并且需要静态地连入内核映象中(有些基本的功能不一定适合采用动态安装模块)，而所有的程序都在一个文件 xyz.c 中，初始化入口为 xyz_init( )，则只要在同一文件中加上一行“__initcall (xyz_init);”就可以了，并不需要到 do_initcalls( )的代码中去作什么修改。

那么，这样的函数指针都有哪一些呢？我们在这里只能略举数例。一个是 mm/slab.c 中定义的 mm_cpucache_init( )，这是 SMP 结构中对 slab 缓冲块队列的一些优化，需要作为初始化程序来执行(mm/slab.c)。

```
471         __initcall(kmem_cpucache_init);
```

经过 gcc 的预处理，就会在初始化调用段中生成出一个指向 kmem_cpucache_init( )的函数指针__initcall_kmem_cpucache_init：

```
static initcall_t __initcall_kmem_cpucache_init
    __attribute__ ((unused,__section__ (".initcall.init"))) = kmem_cpucache_init;
```

再如drivers/pci/proc.c中定义的pci_proc_init( )，这是对特殊文件系统/proc中子目录/proc/pci的初始化，用于PCI总线的管理。

```
437         __initcall(pci_proc_init);
```

经过 gcc 的预处理就会生成一个指向 pci_proc_init( )的函数指针__initcall_pci_proc_init。不过，这些都不是我们所关心的。我们在这里关心的是在 fs/partitions/check.c 中的一个函数 partition_setup( )：

```
456         __initcall(partition_setup);
```

不知道代码的作者为什么给它取了这么个名字，其实这个函数与磁盘的分区毫无关系，实际上处理的是外部设备的初始化，这是系统初始化的一个重要组成部分。

[init( ) > do_basic_setup( ) > do_initcalls( ) > partition_setup( )]

```
442     int __init partition_setup(void)
443     {
444         device_init( );
445
446     #ifdef CONFIG_BLK_DEV_RAM
```

```
447 #ifdef CONFIG_BLK_DEV_INITRD
448         if (initrd_start && mount_initrd) initrd load( );
449         else
450 #endif
451         rd_load( );
452 #endif
453         return 0;
454 }
```

函数的主体(我们忽略条件编译部分)是 device_init( )，其代码在 drivers/block/genhd.c 中：

[init( ) > do_basic_setup( ) > do_initcalls( ) > partition_setup( ) > device_init( )]

```
34  void __init device_init(void)
35  {
36  #ifdef CONFIG_PARPORT
37          parport_init( );
38  #endif
39          chr_dev_init( );
40          blk_dev_init( );
41          sti( );
42  #ifdef CONFIG_I2O
43          i2o_init( );
44  #endif
45  #ifdef CONFIG_BLK_DEV_DAC960
46          DAC960_Initialize( );
47  #endif
48  #ifdef CONFIG_FC4_SOC
49          /* This has to be done before scsi_dev_init */
50          soc_probe( );
51  #endif
52  #ifdef CONFIG_IEEE1394
53                  ieee1394_init( );
54  #endif
55  #ifdef CONFIG_BLK_CPQ_DA
56          cpqarray_init( );
57  #endif
58  #ifdef CONFIG_NET
59          net dev_init( );
60  #endif
61  #ifdef CONFIG_ATM
62          (void) atmdev_init( );
63  #endif
64  #ifdef CONFIG_VT
65          console_map init( );
66  #endif
67  }
```

可见，这是所有外部设备初始化的总入口，里面包括了块设备的初始化 blk_dev_init( )、一般字符设备的初始化 chr_dev_init( )，还有一些特殊设备的初始化，如并行口的 parport_init( )、网络设备的 net_dev_init( )和 atmdev_init( )等等。其中的有些初始化程序已经在“设备驱动”一章中读过，有些(网络设备)不在本书内容的范围之内，有些则虽在本书内容的范围之内但尚未读过。但是，限于本书的篇幅，我们别无选择，只好把这些函数留给读者了。特别地，我们建议读者结合“设备驱动”一章读一下 blk_dev_init( )中调用的 ide_init( )，那是对 IDE 硬盘的初始化。

事实上，所有静态模块的初始化都是通过__initcall( )说明的，只不过是间接的，通过另一个宏定义 module_init( )转了个弯而已(见 include/linux/init.h)：

```
89      #define module_init(x)   __initcall(x);
```

例如，对 Ext2 文件系统的支持就是以模块的形式实现的，所以在 fs/ext2/super.c 中就有这么一行：

```
800     module_init(init_ext2_fs)
```

所以函数 init_ext2_fs( )也是由 do_initcalls( )调用的，实际上其代码也在初始化代码段中：

[init( ) > do_basic_setup( ) > do_initcalls( ) > init_ext2_fs( )]

```
788     static int __init init_ext2_fs(void)
789     {
790             return register_filesystem(&ext2_fs_type);
791     }
```

显然，到 do_initcalls( )执行完毕的时候，所有此类初始化函数都已经执行过了，内核中由静态模块支持的所有文件系统也都已向系统登记。

回到 do_basic_setup( )的代码中，下面(726 行)是对 filesystem_setup( )的调用。不过，这里所谓“文件系统”只是指几个特殊文件系统，主要是 devfs。其他还有作为条件编译选择项的“网络文件系统” nfs 以及对 Unix96 PTY“伪终端”的支持，这些特殊文件系统的安装并不依赖于文件系统的根设备。其中 devfs 的安装是在安装根设备之前的准备工作，如果系统支持 devfs，则 devfs 的初始安装对于在引导命令行中采用“root=”选择项有帮助。不过，这里对 devfs 的安装只是“预安装”，就是通过 kern_mount( )进行的安装，以后还要通过 do_mount( )再“重安装”一次。

[init( ) > do_basic_setup( ) > filesystem_setup( )]

```
28      void __init filesystem_setup(void)
29      {
30          init_devfs_fs( );  /*  Header file may make this empty  */
31
32      #ifdef CONFIG_NFS_FS
33          init_nfs_fs( );
34      #endif
35
36      #ifdef CONFIG_DEVPTS_FS
```

```
37          init_devpts_fs( );
38      #endif
39      }
```

现在到了安装文件系统根设备的时候了，根设备的安装不同于此后其他设备的安装。一般设备(文件系统)的安装都要先通过/dev 目录找到代表着待安装设备的节点，从而取得待安装设备的设备号。而安装点一般(除一些特殊文件系统外)也是在已安装文件系统中的某个节点。安装根设备时/dev 目录尚不存在，但是一般根设备就是引导设备，其设备号已经在参数块中传递了过来，而根设备的安装点“/”本来就不在已经安装的文件系统中，不需要寻找。不过，如果在引导命令行中指定了以另一个设备作为根设备，那就需要一些起码的根据设备名找到设备号的能力，这就是在此之前先预安装 devfs 的目的。此外还有一点不同之处，常规的安装都事先知道设备上是哪一种文件系统(见 do_mount( )的参数)，而根设备的安装却并不事先知道文件系统的类型，而需要在安装时试凑。所以，内核中有个函数 mount_root( )专用于根设备的安装，其代码在 fs/super.c 中。这个函数比较长，所以我们分段阅读，同时我们也从代码中删去了一些条件编译的片段。

[init( ) > do_basic_setup( ) > mount_root( )]

```
1462    void __init mount_root(void)
1463    {
1464        struct file_system_type * fs_type;
1465        struct super_block * sb;
1466        struct vfsmount *vfsmnt;
1467        struct block_device *bdev = NULL;
1468        mode_t mode;
1469        int retval;
1470        void *handle;
1471        char path[64];
1472        int path_start = -1;
1473
1474    #ifdef CONFIG_ROOT_NFS
        . . . . . .
1507    #endif
1508
1509    #ifdef CONFIG_BLK_DEV_FD
        . . . . . .
1529    #endif
1530
1531        devfs_make_root (root_device_name);
1532        handle = devfs_find_handle (NULL, ROOT_DEVICE_NAME,
1533                                    MAJOR (ROOT_DEV), MINOR (ROOT_DEV),
1534                        DEVFS_SPECIAL_BLK, 1);
1535        if (handle)  /*  Sigh: bd*( ) functions only paper over the cracks  */
1536        {
1537            unsigned major, minor;
1538
1539            devfs_get_maj_min (handle, &major, &minor);
```

```
1540            ROOT_DEV = MKDEV (major, minor);
1541        }
```

代码中的 root_device_name[ ]是个全局量，如果在引导命令行中使用了“root=”选择项，就会在一个初始化函数 root_dev_setup( )中从这个选择项中将设备名的前缀“/dev/” 去掉，然后复制到这个字符串中。函数 devfs_make_root( )把旧式的设备名如“ide/hd/0”等等转换成新式的设备名，其一般形式为“/host0/bus1/target2/lun3/part4”。然后，便通过 devfs_find_handle( )在 devfs 文件系统中寻找，这里用到了一些宏操作定义或常数定义(见 fs/super.c 及 include/linux/devfs_fs_kernel.h)：

```
  48    #ifdef CONFIG_BLK_DEV_INITRD
  49    #  define ROOT_DEVICE_NAME((real_root_dev ==ROOT_DEV) ? root_device_name:NULL)
  50    #else
  51    #define ROOT_DEVICE_NAME  root_device_name
  52    #endif

  59    kdev_t ROOT_DEV;
```

ROOT_DEV 看起来像是个常数，实际上却是个全局变量。这个变量的初值是在系统初始化的第一阶段在 setup_arch( )中设置的，实际上来自引导辅助程序和 BIOS。一般而言，从什么设备上引导，那个设备就是根设备，除非通过“root=”选择项另加指定。读者也许感到困惑：系统是由 BIOS 引导的，而“设备号”是 Unix/Linux 所特有的，就算 BIOS 从 IDE 接口上的某个磁盘或盘区引导，它又怎么能知道这个设备在 Linux 中的设备号是什么呢？其实，BIOS (或者 LILO) 只是从引导设备上读入了引导扇区，并且把引导扇区在内存中的映象当作可执行代码而跳转到它的开头处，以后的事就取决于引导扇区的内容和辅助程序 setup 了。而这二者是与具体的系统密切相关的，实际上就是系统的一部分，而且就存放在具体的设备上，当然就能知道这设备的设备号是什么了。

所以，代码中的 1531～1541 行是为通过“root=”选择项改变根设备而设的。如果在引导命令行中没有使用这个选择项，或者内核不支持 devfs，那就保持 ROOT_DEV 的初值不变。

[init( ) > do_basic_setup( ) > mount_root( )]

```
1542
1543        /*
1544         * Probably pure paranoia, but I'm less than happy about delving into
1545         * devfs crap and checking it right now. Later.
1546         */
1547        if (!ROOT_DEV)
1548            panic("I have no root and I want to scream");
1549
1550        bdev = bdget(kdev_t_to_nr(ROOT_DEV));
1551        if (!bdev)
1552            panic(__FUNCTION__ ": unable to allocate root device");
1553        bdev->bd_op = devfs_get_ops (handle);
1554        path_start = devfs_generate_path (handle, path + 5, sizeof (path) - 5);
1555        mode = FMODE_READ;
```

```
1556        if (!(root_mountflags & MS_RDONLY))
1557            mode |= FMODE_WRITE;
1558        retval = blkdev_get(bdev, mode, 0, BDEV_FS);
1559        if (retval == -EROFS) {
1560            root_mountflags |= MS_RDONLY;
1561            retval = blkdev_get(bdev, FMODE_READ, 0, BDEV_FS);
1562        }
1563        if (retval) {
1564                /*
1565             * Allow the user to distinguish between failed open
1566             * and bad superblock on root device.
1567             */
1568            printk ("VFS: Cannot open root device \"%s\" or %s\n",
1569                root_device_name, kdevname (ROOT_DEV));
1570            printk ("Please append a correct \"root=\" boot option\n");
1571            panic("VFS: Unable to mount root fs on %s",
1572                kdevname(ROOT_DEV));
1573        }
1574
1575        check_disk_change(ROOT_DEV);
1576        sb = get_super(ROOT_DEV);
1577        if (sb) {
1578            fs_type = sb->s_type;
1579            goto mount_it;
1580        }
1581
```

知道了根设备的设备号，下面的操作就与普通的安装没有多大不同了，因为设备驱动层已经在此之前完成了初始化，根据设备号就可以找到有关的数据结构。例如，1550 行的 bdget( )根据设备号找到或者创建给定设备的 block_device 数据结构，读者已经在“设备驱动”一章中读过它的代码。1553 行的 devfs_get_ops( )找到设备的 block_device_operations 数据结构。然后，blkdev_get( )通过这个数据结构中提供的 open 操作“打开”目标设备。这里先在 1558 行试着按可写模式打开，如果失败就在 1561 行再按只读模式打开。至于 1576 行的 get_super( )，则扫描已经读入系统中的超级块队列。如果在这个队列中找到了目标设备上的超级块，就可以知道该设备上的文件系统是什么类型的(1578 行)，从而可以进入真正的“安装”操作(1579 行)。当然，对于根设备的安装，其超级块还不在超级块队列中，因而需要从设备上读入。

我们接着往下看。

[init( ) > do_basic_setup( ) > mount_root( )]

```
1582        read_lock(&file_systems_lock);
1583        for (fs_type = file_systems ; fs_type ; fs_type = fs_type->next) {
1584            if (!(fs_type->fs_flags & FS_REQUIRES_DEV))
1585                continue;
1586            if (!try_inc_mod_count(fs_type->owner))
1587                continue;
```

```
1588            read_unlock(&file_systems_lock);
1589            sb = read_super(ROOT_DEV,bdev,fs_type,root_mountflags,NULL,1);
1590            if (sb)
1591                goto mount_it;
1592            read_lock(&file_systems_lock);
1593            put_filesystem(fs_type);
1594        }
1595        read_unlock(&file_systems_lock);
1596        panic("VFS: Unable to mount root fs on %s", kdevname(ROOT_DEV));
1597
```

内核中的 file_systems 是一个 file_system_type 数据结构队列，队列中的每个数据结构都代表着一种受到内核支持的文件系统。队列中的各个数据结构都通过函数指针提供其自己的 read_super 操作。这里依次以各种文件系统的 file_system_type 数据结构为参数调用 read_super( )，实际上就是让各种文件系统都来试试，看谁能从设备上读入一个格式相符的超级块。读者已经在“文件系统”一章中读过 read_super( )的代码。

读入了超级块，就可以进入安装阶段了。

[init( ) > do_basic_setup( ) > mount_root( )]

```
1598    mount_it:
1599        printk ("VFS: Mounted root (%s filesystem)%s.\n",
1600            fs_type->name,
1601            (sb->s_flags & MS_RDONLY) ? " readonly" : "");
1602        if (path_start >= 0) {
1603            devfs_mk_symlink (NULL, "root", DEVFS_FL_DEFAULT,
1604                        path + 5 + path_start, NULL, NULL);
1605            memcpy (path + path_start, "/dev/", 5);
1606            vfsmnt = add_vfsmnt(NULL, sb->s_root, path + path_start);
1607        }
1608        else
1609            vfsmnt = add_vfsmnt(NULL, sb->s_root, "/dev/root");
1610        /* FIXME: if something will try to umount us right now... */
1611        if (vfsmnt) {
1612            set_fs_root(current->fs, vfsmnt, sb->s_root);
1613            set_fs_pwd(current->fs, vfsmnt, sb->s_root);
1614            if (bdev)
1615                bdput(bdev); /* sb holds a reference */
1616            return;
1617        }
1618        panic("VFS: add_vfsmnt failed for root fs");
1619    }
```

函数 add_vfsmnt( )的代码已经在“文件系统”一章中读过，这里就不多说了。函数 set_fs_root( )设置当前进程的 fs_struct 数据结构中的 root 和 rootmnt 两个指针，使它们分别指向根节点的 dentry 数据结构，就是这里的 sb->s_root，以及安装根设备时的“连接件” vfsmount 数据结构，就是这里的 vfsmnt。

同时，set_fs_pwd( )将当前进程的“当前工作目录”，即 pwd 和 pwdmnt 两个指针也设置成指向这两个数据结构。读者将会看到，现在的这个当前进程就是系统中所有进程的始祖，把这个进程的这些指针设置好了，以后由它 fork( )出来的进程就都会继承这些指针，以整个文件系统的根为本进程逻辑意义上的“根”，并且以此为本进程的当前工作目录。

完成了根设备的安装以后，如果需要的话，就可以进一步完成特殊文件系统 devfs 的安装。在安装根设备之前，已经通过 kern_mount( )将 devfs 预安装了一次。但是，我们以前提到过，有些文件系统需要在此后再“重安装”一次，这样才能与具体文件系统中的某个具体的节点挂上钩。所以 do_basic_setup( )接着就调用 mount_devfs_fs( )来做这件事(fs/devfs/base.c)。

[init( ) > do_basic_setup( ) > mount_devfs_fs( )]

```
3363    void __init mount_devfs_fs (void)
3364    {
3365        int err;
3366
3367        if ( (boot_options & OPTION_NOMOUNT) ) return;
3368        err = do_mount ("none", "/dev", "devfs", 0, "");
3369        if (err == 0) printk ("Mounted devfs on /dev\n");
3370        else printk ("Warning: unable to mount devfs, err: %d\n", err);
3371    }   /*  End Function mount_devfs_fs  */
```

可见，如果内核支持 devfs，并且需要安装，那就是把它安装在/dev 节点上。

至此，do_basic_setup( )的执行已经完成。回到 init( )的代码中，初始化代码段中的函数都已经执行过了。如前所述，这些函数都只需要在系统初始化时执行一次，现在既然已经完成，就可以“过河拆桥”，回收它们所占的空间另作他用了。函数 free_initmem( )的作用就是逐个页面地回收整个初始化代码段的内存空间，其代码在 mm/init.c 中：

[init( ) > free_initmem( )]

```
656    void free_initmem(void)
657    {
658        unsigned long addr;
659
660        addr = (unsigned long)(&__init_begin);
661        for (; addr < (unsigned long)(&__init_end); addr += PAGE_SIZE) {
662            ClearPageReserved(virt_to_page(addr));
663            set_page_count(virt_to_page(addr), 1);
664            free_page(addr);
665            totalram_pages++;
666        }
667      printk ("Freeing unused kernel memory: %dk freed\n",
                                          (&__init_end - &__init_begin) >> 10);
668    }
```

至此，内核的初始化已经完成，下面要再往上跑一层，处理应用层的初始化，为应用程序的运行，

首先是系统管理作好准备了。我们接着往下看 init( )的代码。

[init( )]

```
774         if (open("/dev/console", O_RDWR, 0) < 0)
775             printk("Warning: unable to open an initial console.\n");
776 
777         (void) dup(0);
778         (void) dup(0);
779 
780         /*
781          * We try each of these until one succeeds.
782          *
783          * The Bourne shell can be used instead of init if we are
784          * trying to recover a really broken machine.
785          */
786 
787         if (execute_command)
788             execve(execute_command,argv_init,envp_init);
789         execve("/sbin/init",argv_init,envp_init);
790         execve("/etc/init",argv_init,envp_init);
791         execve("/bin/init",argv_init,envp_init);
792         execve("/bin/sh",argv_init,envp_init);
793         panic("No init found.  Try passing init= option to kernel.");
794     }
```

这里的 dup( )和 execve( )都是系统调用。当前进程 init( )是个运行于系统空间的内核线程，虽然也能作系统调用，却不能像在用户空间那样通过普通的 C 语言库函数进行，而要由内核中的系统调用入口函数，我们以 execve( )为例来看这些函数的定义(include/asm-i386/ unistd.h)：

```
273     #define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) \
274     type name(type1 arg1,type2 arg2,type3 arg3) \
275     { \
276     long __res; \
277     __asm__ volatile ("int $0x80" \
278         : "=a" (__res) \
279         : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
280               "d" ((long)(arg3))); \
281     __syscall return(type,__res); \
282     }

233     #define __syscall_return(type, res) \
234     do { \
235         if ((unsigned long)(res) >= (unsigned long)(-125)) { \
236             errno = -(res); \
237             res = -1; \
238         } \
```

```
239        return (type) (res); \
240    } while (0)
```

宏操作_syscall3( )用于所有带三个参数的系统调用，而 execve( )就是这样的系统调用：

```
342    static inline _syscall3(int, execve, const char *, file,
                                 char **, argv, char **, envp)
```

经过 gcc 的预处理以后，就生成了函数 execve( )的定义：

```
int execve(const char * file, char **argv, char **envp)
{
  long   res;
  __asm__ volatile ("int $0x80"
      : "=a" (__res)
      : "0" (__NR_execve),"b" ((long)(file)),"c" ((long)(argv)),
         "d" ((long)(envp)));

   do {
        if ((unsigned long)(res) >= (unsigned long)(-125)) {
            errno = -(res);
            res = -1;
        }
        return (int) (res);
  } while (0)
}
```

对这段代码可能需要一些解释。这里的输入部说明把参数 file 放在寄存器%ebx 中，把参数 argv 放在寄存器 %ecx 中，把参数 envp 放在寄存器 %edx 中。系统调用号__NR_execve 则放在操作数 0 中，那就是局部变量 res。由于 res 是局部量，所以是在堆栈中，而且在这里是在堆栈顶部。所有这些正好构成了执行指令“int 0x80”进入系统的条件。同时，输入部又说明 res 既是局部变量(因而在堆栈中)，又跟寄存器%eax 结合，因为系统调用通过%eax 返回执行的结果。

需要在内核中调用的其他系统调用，如 dup( )、open( )等等，都是按同样的方式生成的。内核中相应地还有用于不带参数、带 1 个参数、带 2 个参数等等的宏操作_syscall0( )、_syscall1( )、_syscall2( )等等。

前面的代码中先打开/dev/console，由于这是当前进程第一次打开文件，其打开文件号必定为 0。接着调用 dup( )把由打开文件号 0 所代表的连接复制两次，就使打开文件号 0、1 和 2 都成为同一个连接的代表，这就是通常所说的“标准输入”、“标准输出”以及“标准出错信息”三个通道 stdin、stdout 以及 stderr。打开了这三个标准 I/O 通道以后，就可以像普通在 shell 下启动的进程那样，通过系统调用 execve( )执行各种可执行程序了。读者在第 4 章中看到，execve( )是条不归之路，一旦执行目标程序成功，最后就从目标程序直接 exit( )了。但是，如果执行目标程序失败，例如在文件系统中找不到目标程序，那就会从 execve( )失败返回。所以，这里实际上是依次尝试执行/sbin/init、/etc/init、/bin/init、/bin/sh。一般来说，至少/bin/sh 是一定可以执行成功的，否则系统就根本无法运行了。

那么，/sbin/init 又干些什么呢？主要是根据文件/etc/inittab 的规定分叉(创建)出一些进一步初始化

的进程，包括对/etc/rc.d/rc 的执行，不过那已经不属于内核的范围了，读者可以通过命令“man 8 init”阅读对这个程序的说明，或阅读有关 Linux 系统管理的资料。分叉出来的进程中还包括若干执行/sbin/getty 的进程，这些进程后来就变成了执行/bin/login 或/sbin/sulogin 的进程。就这样，这个以系统初始化为开始的进程就成为整个系统中所有进程共同的“祖宗”，这个进程以后一直留在系统中，就是系统的 1 号进程。

总之，系统中有几个处理器就有几个 0 号进程，那就是各个处理器的空转进程，这些进程都不在进程的杂凑队列中，也不在就绪进程队列中，不参与进程调度。但是，1 号进程却只有一个，这是系统中所有进程的祖宗，从这个进程开始的所有进程都参与调度。

10.5 系统的关闭和重引导

最后我们再看看关闭系统的过程。Linux 为此提供了一个系统调用 reboot( )，内核中的实现为 sys_reboot( )。顾名思义，这个系统调用的功能应该是“重引导”，但是实际上这是个多功能的系统调用。根据参数 cmd 的值，这个系统调用可以用于以下一些目的：

- “重引导” (LINUX_REBOOT_CMD_RESTART 或 LINUX_REBOOT_CMD_RESTART2)。
- “停机” (LINUX_REBOOT_CMD_HALT)。
- “关机” (LINUX_REBOOT_CMD_POWER_OFF)。

此外，还可以用来设置键组 C_A_D，即 Ctrl-Alter-Delete 三键同按时的作用。在 DOS 和 Windows 操作系统上同时按这三个键表示重引导（在 Windows NT 上则又不是），所以 Linux 允许用户选择用或不用这个特殊的组合。

在为重引导或停机而调用 reboot( )之前，应用进程应该先调用另一个系统调用 sync( )，以保证把文件系统中所有已经改变了的缓冲区和数据结构的内容先写入磁盘。

函数 sys_reboot( )的代码在 kernel/sys.c 中：

```
261    /*
262     * Reboot system call: for obvious reasons only root may call it,
263     * and even root needs to set up some magic numbers in the registers
264     * so that some mistake won't make this reboot the whole machine.
265     * You can also set the meaning of the ctrl-alt-del-key here.
266     *
267     * reboot doesn't sync: do that yourself before calling this.
268     */
269    asmlinkage long sys_reboot(int magic1, int magic2, unsigned int cmd, void * arg)
270    {
271        char buffer[256];
272
273        /* We only trust the superuser with rebooting the system. */
274        if (!capable(CAP_SYS_BOOT))
275            return -EPERM;
276
277        /* For safety, we require "magic" arguments. */
278        if (magic1 != LINUX_REBOOT_MAGIC1 ||
```

```
279             (magic2 != LINUX_REBOOT_MAGIC2 && magic2 != LINUX_REBOOT_MAGIC2A &&
280                 magic2 != LINUX_REBOOT_MAGIC2B))
281             return -EINVAL;
282
283         lock_kernel( );
284         switch (cmd) {
285         case LINUX_REBOOT_CMD_RESTART:
286             notifier_call_chain(&reboot_notifier_list, SYS_RESTART, NULL);
287             printk(KERN_EMERG "Restarting system.\n");
288             machine_restart(NULL);
289             break;
290
291         case LINUX_REBOOT_CMD_CAD_ON:
292             C_A_D = 1;
293             break;
294
295         case LINUX_REBOOT_CMD_CAD_OFF:
296             C_A_D = 0;
297             break;
298
299         case LINUX_REBOOT_CMD_HALT:
300             notifier_call_chain(&reboot_notifier_list, SYS_HALT, NULL);
301             printk(KERN_EMERG "System halted.\n");
302             machine_halt( );
303             do_exit(0);
304             break;
305
306         case LINUX_REBOOT_CMD_POWER_OFF:
307             notifier_call_chain(&reboot_notifier_list, SYS_POWER_OFF, NULL);
308             printk(KERN_EMERG "Power down.\n");
309             machine_power_off( );
310             do_exit(0);
311             break;
312
313         case LINUX_REBOOT_CMD_RESTART2:
314             if (strncpy_from_user(&buffer[0], (char *)arg, sizeof(buffer)-1) < 0) {
315                 unlock_kernel( );
316                 return -EFAULT;
317             }
318             buffer[sizeof(buffer) - 1] = '\0';
319
320             notifier_call_chain(&reboot_notifier_list, SYS_RESTART, buffer);
321             printk(KERN_EMERG "Restarting system with command '%s'.\n", buffer);
322             machine_restart(buffer);
323             break;
324
325         default:
326             unlock_kernel( );
```

```
327            return -EINVAL;
328        }
329        unlock kernel( );
330        return 0;
331    }
```

在 sys_reboot( )所实现的这些功能中，以重引导最为复杂，所以我们在这里读一下与此有关的代码，其余的就留给读者了。前面讲过，系统引导时可以使用一个带选择项的命令行，也可以不使用命令行，所以 sys_reboot( )也为重引导提供了两个不同的命令码。我们只看不用命令行的情景，就是代码中的 286~289 行。

内核中的有些模块(通常是外部设备)可能要求在关闭系统或重引导之前执行一些特殊的操作，例如有些老的硬盘就要求在断电之前先把磁头移到一个特定的“起降”磁道上。所以，要提供一种手段，使这样的模块在关闭系统或重引导之前能得到通知，执行一个预定的函数。这就是代码中调用 notifier_call_chain( )的目的。如果一个模块要求在关闭系统或重引导之前执行一个函数，就可以准备好一个notifier_block数据结构，并向系统登记。这种数据结构的定义在include/linux/notifier.h中：

```
14    struct notifier_block
15    {
16        int (*notifier_call)(struct notifier_block *self, unsigned long, void *);
17        struct notifier_block *next;
18        int priority;
19    };
```

结构中的函数指针就用于需要在关闭系统或重引导之前执行的函数。准备好 notifier_block 数据结构以后，就可以通过 register_reboot_notifier( )向系统登记(kernel/sys.c)：

```
149    int register_reboot_notifier(struct notifier_block * nb)
150    {
151        return notifier chain_register(&reboot_notifier_list, nb);
152    }
```

在 sys_reboot( )中，当要关闭系统或重引导系统的时候，就通过 notifier_call_chain( )执行已经登记的函数，调用的参数之一是一个表示调用原因的代码。

[sys_reboot( ) > notifier_call_chain( )]

```
105    /**
106     *  notifier_call chain - Call functions in a notifier chain
107     *  @n: Pointer to root pointer of notifier chain
108     * @val: Value passed unmodified to notifier function
109     *   @v: Pointer passed unmodified to notifier function
110     *
111     *  Calls each function in a notifier chain in turn.
112     *
113     *  If the return value of the notifier can be and'd
```

```
114      * with %NOTIFY_STOP_MASK, then notifier call_chain
115      *   will return immediately, with the return value of
116      *   the notifier function which halted execution.
117      *   Otherwise, the return value is the return value
118      *   of the last notifier function called.
119      */
120
121     int notifier_call_chain(struct notifier_block **n, unsigned long val, void *v)
122     {
123         int ret=NOTIFY_DONE;
124         struct notifier_block *nb = *n;
125
126         while(nb)
127         {
128             ret=nb->notifier_call(nb,val,v);
129             if(ret&NOTIFY_STOP_MASK)
130             {
131                 return ret;
132             }
133             nb=nb->next;
134         }
135         return ret;
136     }
```

调用了已经登记的函数以后，就可以通过 machine_restart( )重引导了，这个函数的代码在 arch/i386/kernel/process.c 中：

[sys_reboot( ) > machine_restart( )]

```
346     void machine_restart(char *    unused)
347     {
348     #if CONFIG_SMP
349         /*
350          * Stop all CPUs and turn off local APICs and the IO-APIC, so
351          * other OSs see a clean IRQ state.
352          */
353         smp_send_stop( );
354         disable_IO_APIC( );
355     #endif
356
357         if(!reboot_thru_bios) {
358             /* rebooting needs to touch the page at absolute addr 0 */
359             *((unsigned short *)__va(0x472)) = reboot_mode;
360             for (;;) {
361                 int i;
362                 for (i=0; i<100; i++) {
363                     kb_wait( );
364                     udelay(50);
```

```
365                     outb(0xfe,0x64);         /* pulse reset low */
366                     udelay(50);
367                 }
368                 /* That didn't work - force a triple fault.. */
369                 __asm__ __volatile__("lidt %0": :"m" (no_idt));
370                 __asm__ __volatile__("int3");
371             }
372         }
373
374         machine_real_restart(jump_to_bios, sizeof(jump_to_bios));
375     }
```

对于SMP结构的系统，先要通过smp_send_stop( )向系统中的其他CPU发送一个请求停机的处理器间中断请求，并关闭本CPU的“高级中断控制器”APIC。此外，执行sys_reboot( )的CPU还负有关闭全局的外部APIC的责任。然后，就要进行实际的“重启动”，即重引导了。

对系统的重启动可以通过两种不同的方法进行，一种是通过BIOS进行，另一种就是直接在系统中制造一次“总清”(reset)。具体采用哪一种方式取决于系统引导时是否使用了“root=”选择项。如果没有使用这个选择项，那么全局量reboot_thru_bios为0，所以会采用所谓“硬启动”制造一次总清。具体的过程见360~371行，我们就不深入到这些细节中去了，有兴趣或需要的读者可以结合PC和i386的有关技术资料自行阅读。

如果在引导命令行中使用了“root=”选择项，则系统会在初始化的过程中执行一个函数reboot_setup( )，根据选择项的内容设置reboot_thru_bios和reboot_mode两个全局量的值，作为执行reboot( )系统调用时的依据。

```
157     static int __init reboot_setup(char *str)
158     {
159         while(1) {
160             switch (*str) {
161             case 'w': /* "warm" reboot (no memory testing etc) */
162                 reboot_mode = 0x1234;
163                 break;
164             case 'c': /* "cold" reboot (with memory testing etc) */
165                 reboot_mode = 0x0;
166                 break;
167             case 'b': /* "bios" reboot by jumping through the BIOS */
168                 reboot_thru_bios = 1;
169                 break;
170            case 'h': /* "hard" reboot by toggling RESET and/or crashing the CPU */
171                 reboot_thru_bios = 0;
172                 break;
173             }
174             if((str = strchr(str,',')) != NULL)
175                 str++;
176             else
177                 break;
```

```
178         }
179         return 1;
180     }
181
182     __setup("reboot=", reboot_setup);
```

如果在命令行中选择了通过 BIOS 重引导，则又有“热引导”和“冷引导”之分，二者都通过 machine_real_restart( )进行。然而，不管是“热引导”还是“冷引导”，总之都要进入 BIOS。这里所谓进入 BIOS 并不是指像“int 0x13”那样的 BIOS 调用，而是要进入整个 BIOS 的初始入口，就好像机器刚加电一样。以前我们提到过，BIOS 的入口是 0xffff0，读者可能觉得这很容易，只要执行一条 jmp 指令跳转到这个地址就行了。可是，事情并不是这么简单，BIOS 是为 16 位实地址模式设计的，而 CPU 此刻运行于 32 位保护模式，而且采用页式地址映射，这中间有着不小的差距。当初，次 CPU 在受到启动后是经过了一段“跳板”程序才从实地址模式进入保护模式；而主 CPU 则先执行了一段引导辅助程序，实际上也是“跳板”，才进入保护模式的。而且，二者在进入 startup_32( )以后又经历了向页式映射的过渡。现在要回到 BIOS 就得走过相反的过程，上台靠跳板，下台要有台阶。而 machine_real_restart( )的作用正是让当前 CPU 通过台阶下台。这个函数的代码在 kernel/process.c 中：

[sys_reboot( ) > machine_restart( ) > machine_real_restart( )]

```
254     /*
255      * Switch to real mode and then execute the code
256      * specified by the code and length parameters.
257      * We assume that length will aways be less that 100!
258      */
259     void machine_real_restart(unsigned char *code, int length)
260     {
261         unsigned long flags;
262
263         cli( );
264
265         /* Write zero to CMOS register number 0x0f, which the BIOS POST
266            routine will recognize as telling it to do a proper reboot.  (Well
267            that's what this book in front of me says -- it may only apply to
268            the Phoenix BIOS though, it's not clear).  At the same time,
269            disable NMIs by setting the top bit in the CMOS address register,
270            as we're about to do peculiar things to the CPU.  I'm not sure if
271            `outb_p' is needed instead of just `outb'.  Use it to be on the
272            safe side.  (Yes, CMOS_WRITE does outb_p's. -  Paul G.)
273          */
274
275         spin_lock_irqsave(&rtc_lock, flags);
276         CMOS_WRITE(0x00, 0x8f);
277         spin_unlock_irqrestore(&rtc_lock, flags);
278
279         /* Remap the kernel at virtual address zero, as well as offset zero
280            from the kernel segment.  This assumes the kernel segment starts at
```

```
281             virtual address PAGE_OFFSET. */
282
283         memcpy (swapper_pg dir, swapper_pg_dir + USER_PGD_PTRS,
284             sizeof (swapper_pg_dir [0]) * KERNEL_PGD_PTRS);
285
286         /* Make sure the first page is mapped to the start of physical memory.
287            It is normally not mapped, to trap kernel NULL pointer dereferences.*/
288
289         pg0[0] = _PAGE_RW | _PAGE_PRESENT;
290
291         /*
292          * Use `swapper_pg_dir' as our page directory.
293          */
294         asm volatile("movl %0,%%cr3": :"r" ( _pa(swapper_pg dir)));
295
296         /* Write 0x1234 to absolute memory location 0x472.  The BIOS reads
297            this on booting to tell it to "Bypass memory test (also warm
298            boot)".  This seems like a fairly standard thing that gets set by
299            REBOOT.COM programs, and the previous reset routine did this
300            too. */
301
302         *((unsigned short *)0x472) = reboot_mode;
303
304         /* For the switch to real mode, copy some code to low memory.  It has
305            to be in the first 64k because it is running in 16-bit mode, and it
306            has to have the same physical and virtual address, because it turns
307            off paging.  Copy it near the end of the first page, out of the way
308            of BIOS variables. */
309
310         memcpy ((void *) (0x1000 - sizeof (real mode_switch) - 100),
311             real_mode switch, sizeof (real_mode_switch));
312         memcpy ((void *) (0x1000 - 100), code, length);
313
314         /* Set up the IDT for real mode. */
315
316         __asm__ __volatile__ ("lidt %0" : : "m" (real_mode_idt));
317
318         /* Set up a GDT from which we can load segment descriptors for real
319            mode.  The GDT is not used in real mode; it is just needed here to
320            prepare the descriptors. */
321
322         __asm__ __volatile__ ("lgdt %0" : : "m" (real_mode gdt));
323
324         /* Load the data segment registers, and thus the descriptors ready for
325            real mode.  The base address of each segment is 0x100, 16 times the
326            selector value being loaded here.  This is so that the segment
327            registers don't have to be reloaded after switching to real mode:
328            the values are consistent for real mode operation already. */
```

```
329
330         __asm__ __volatile__ ("movl $0x0010,%%eax\n"
331                     "\tmovl %%eax,%%ds\n"
332                     "\tmovl %%eax,%%es\n"
333                     "\tmovl %%eax,%%fs\n"
334                     "\tmovl %%eax,%%gs\n"
335                     "\tmovl %%eax,%%ss" : : : "eax");
336
337         /* Jump to the 16-bit code that we copied earlier.  It disables paging
338            and the cache, switches to real mode, and jumps to the BIOS reset
339            entry point. */
340
341         __asm__ __volatile__ ("ljmp $0x0008,%0"
342                 :
343                 : "i" ((void *)(0x1000 - sizeof (real_mode_switch) - 100)));
344     }
```

整个过渡的过程都不容许中断，所以一开始先关闭中断。然后向 CMOS 存储器中地址为 0x8f 处写入 0。注释中说这是 BIOS 所要求的，并且同时也起了关闭“不可屏蔽中断”NMI 的作用，我们就不加考证了。

```
22    #define CMOS_WRITE(val, addr) ({ \
23    outb_p((addr),RTC_PORT(0)); \
24    outb_p((val),RTC_PORT(1)); \
25    })
```

以前，在向页式映射的过渡中，有一个时期需要在页面目录中的低区，即从虚地址 0 开始的区间也保持部分空间的映射，使得 CPU 以虚拟地址和物理地址访问内存时可以被映射到相同的物理存储单元。现在要向相反方向过渡也得要有这么一个时期，所以 283 行从页面映射目录 swapper_pg_dir 中把对应于系统空间的 256 个目录项复制到低区，放在从目录起点开始的地方。283～284 行中引用的常数分别定义于 include/asm-i386/pgtable-2level.h 和 include/asm-i386/pgtable.h 中：

```
  8    #define PGDIR_SHIFT          22

123    #define USER_PGD_PTRS        (PAGE_OFFSET >> PGDIR_SHIFT)
124    #define KERNEL_PGD_PTRS      (PTRS_PER_PGD-USER_PGD_PTRS)
```

这些目录项是从页面映射目录的高区原封不动复制下来的，可是系统空间的第一个页面、即 pg0[0] 在正常运行中是无映射的，所以复制以后还要补上一个(见 289 行)。这里标志位_PAGE_RW 和_PAGE_PRESENT 的作用不言自明，而页面的基地址则为 0。实际上，这个页面正是现在要用的。

由于页面映射目录已经改变，294 行再装入一次控制寄存器%cr3。当然，因为是在内核中运行，原来%cr3 也是指向 swapper_pg_dir，装入前后这个地址并无改变，但是这条指令的执行使 CPU 重新装入页面映射目录。

如前所述，全局量 reboot_mode 的值是(当初)根据引导命令行中的选择项设置的。如果在命令行中选择了“热引导”，那么这个变量的值就是 0x1234（见前面的 162 行)，否则就是 0。BIOS 在执行中要

测试内存单元 0x472 的内容，如果是 0 就要执行对内存等的自检，称为“热引导”；而若为 0 则跳过自检，称为“冷引导”。所以，302 行把 reboot_mode 的值写入这个内存单元，为 BIOS 的运行作好准备。

接下来(310～312 行)，还要把两段小程序复制到物理内存第一个页面的顶部。这两个小程序就起着“台阶”的作用，一个是 real_mode_switch，另一个是作为参数传下来的 jump_to_bios。等一下读者就会看到这两个台阶是如何起作用的。

准备好台阶，就可以下了。首先(316 行)是把中断描述表换成 real_mode_idt，实际上是把所有的中断门和陷阱门都清除掉：

```
205     real_mode_idt = { 0x3ff, 0 };
```

全局段描述表的改变就是关键性的了。322 行将新的全局段描述表 real_mode_gdt 装入 GDTR，其内容定义于 arch/i386/kernel/process.c：

```
184     /* The following code and data reboots the machine by switching to real
185        mode and jumping to the BIOS reset entry point, as if the CPU has
186        really been reset.  The previous version asked the keyboard
187        controller to pulse the CPU reset line, which is more thorough, but
188        doesn't work with at least one type of 486 motherboard.  It is easy
189        to stop this code working; hence the copious comments. */
190
191     static unsigned long long
192     real_mode_gdt_entries [3] =
193     {
194         0x0000000000000000ULL,  /* Null descriptor */
195         0x00009a000000ffffULL,  /* 16-bit real-mode 64k code at 0x00000000 */
196         0x000092000100ffffULL   /* 16-bit real-mode 64k data at 0x00000100 */
197     };
198
199     static struct
200     {
201         unsigned short       size __attribute__ ((packed));
202         unsigned long long * base __attribute__ ((packed));
203     }
204     real_mode_gdt = { sizeof (real_mode_gdt_entries) - 1,
                                                real_mode_gdt_entries },
```

新的段描述项在数组 real_mode_gdt_entries[ ]中。对照第 2 章中对段描述项格式的定义，就可以看出，下标(即段选择号)为 1 的段描述项对应于代码段，其基地址为 0；而下标为 2 的段描述项则对应于数据段，其基地址为 100；两个段的大小都是 64KB。接着(330 行)把除 CS 以外所有的段寄存器都设置成 0x0010，即都是数据段。最后(341～343 行)是一条长程转移指令 ljmp（前缀“l”表示长距离）。跳转的目标怎样确定呢？段选择项是 0x0008，对应于 real_mode_gdt_entries[ ]中的第二项(195 行)，即代码段，所以段的基地址为 0，而位移量则是(0x1000－sizeof (real_mode_switch)－100)。显然，这就是前面复制好的台阶 real_mode_switch 的入口。这段程序以及 jump_to_bios 的机器代码以数据的形式定义于 arch/i386/kernel/process.c 中：

```
226     static unsigned char real_mode_switch [ ] =
227     {
228         0x66, 0x0f, 0x20, 0xc0,             /*    movl   %cr0, %eax          */
229         0x66, 0x83, 0xe0, 0x11,             /*    andl   $0x00000011, %eax */
230         0x66, 0x0d, 0x00, 0x00, 0x00, 0x60, /*    orl    $0x60000000, %eax */
231         0x66, 0x0f, 0x22, 0xc0,             /*    movl   %eax, %cr0          */
232         0x66, 0x0f, 0x22, 0xd8,             /*    movl   %eax, %cr3          */
233         0x66, 0x0f, 0x20, 0xc3,             /*    movl   %cr0, %ebx          */
234         0x66, 0x81, 0xe3, 0x00,
            0x00, 0x00, 0x60,                   /*    andl   $0x60000000, %ebx */
235         0x74, 0x02,                         /*    jz     f                   */
236         0x0f, 0x08,                         /*    invd                       */
237         0x24, 0x10,                         /* f: andb   $0x10, al           */
238         0x66, 0x0f, 0x22, 0xc0              /*    movl   %eax, %cr0          */
239     };
240     static unsigned char jump_to_bios [ ] =
241     {
242         0xea, 0x00, 0x00, 0xff, 0xff        /*    ljmp   $0xffff, $0x0000    */
243     };
```

进入 228 行以后，由于 jmp 指令将与物理地址等同的线性地址装入了 IP，从此开始取指令的地址就都在低区了，这就是为什么需要在页面映射目录中先作好准备的原因。接着，通过改变控制寄存器%cr0 和%cr3 的内容将页面映射关闭。如果 CPU 的高速缓存还开着，则还要执行一条 invd 指令将高速缓存中的内容作废。然后，又通过改变%cr0 的内容回到实地址模式。执行完 238 行的 mov 指令，CPU 已经运行于实地址模式了。程序 jump_to_bios 的机器代码就紧接在 real_mode_switch[ ]的上方，所以执行完 238 行的 mov 指令以后紧接着就是 242 行的(长程)jmp 指令，目标是段地址为 0xfff，位移为 0 的地方，即 0xffff0，这就是 BIOS 的入口。